Python Tkinter By Example

David Love

January 18, 2018

Contents

0.1 Introduction

Thank you for taking an interest in my book. Its purpose is to teach you everything you should need to know to begin using Tkinter in Python 3. Examples in this book cover Tkinter 8.6 in Python 3.6. If you wish to follow along using Python 2, there shouldn't be too many differences, but keep in mind I haven't tested the code for compatability. The main thing to note is that the module is likely called `Tkinter` (capital T), but in Python 3 it is `tkinter` (small t).

Each chapter of this book is written in the form of an image of the target application with the app's entire source code, followed by a breakdown and explanation of the code. Each example is included to teach a specific topic (or a bunch of related ones). Apps are developed iteratively, with each step adding a new feature and teaching a key part. Code which has not changed from the previous iteration will be condensed with ellipses (...) for brevity. I have also included some exercises at the end of each chapter for anyone who wishes to practice development by themselves.

0.2 Who this book is aimed at

This book is written for anyone who knows python and wants to learn a bit about developing GUI applications. Whether you've got a command line application you want to make friendlier with a GUI or you have a great idea for a GUI app which you want to get started on, this book will hopefully give you the tools you need to begin writing your own tkinter apps from scratch.

I will assume that you have basic knowledge of python programming already, and will not explain things like installing python, running programs, or basic syntax (things like `if`, `for` loops and such). At the same time, you will not need to be an expert to follow along either. I would suggest learning about `Classes` if you aren't already aware of them, as all of the examples are written using a class.

I hope you are able to learn something interesting from this book. Should you have any questions, feel free to contact me. I'm @Dvlv292 on Twitter and Dvlv on Reddit.

All source code from this book is freely available on my Github at http://github.com/Dvlv/Tkinter-By-Example.

0.3 How to get the most out of this book

The best way to ensure that the knowledge from any programming book really sticks in your mind is to *write out the code* for yourself. You can do this whilst reading the section or after finishing the explanation; it doesn't really matter. The important thing is that you code along with the book. Reading the code can only get you so far - you need to practise, practise, practise!

Don't just follow along either. If you wonder *"what if I change this"* or *"couldn't I do it like that?"* then just do it! If you mess up, just start again, or grab the code from Github and "reset" back to where you were. You cannot go wrong.

0.4 About tkinter

0.4.1 Installing

Tkinter is probably already installed alongside python. Some Linux distros may not include it, so you might have to look for `python3-tkinter` in your package manager. Check by running python in a terminal and trying to do `>>> import tkinter`.

0.4.2 What is it anyway?

Tkinter is a GUI library. It comes with everything you would need to begin making GUI applications such as buttons, text inputs, radio buttons, dropdowns, and more. Thanks to its inbuilt module ttk it also has the ability to provide some advanced features like tabbed windows, tree views, and progress bars.

0.4.3 Why write about tkinter?

I have an unexplainable attachment to tkinter. I think it was the second python module which I began using for a big project - after pygame - and so I just have some nostalgia towards it. Personal preference aside, since tkinter is built into python as part of the standard library, it's pretty much a go-to for new users who want to try out making a GUI. There are no awkward dependencies, no licence issues, and in my opinion it's very easy to pick up and play with. There are lots of great StackOverflow answers for common problems one may run into and the documentation isn't bad either. I think tkinter is the easiest and best library for those who are new to GUI development. Overall though, I'm writing about tkinter because I like it, and I'm having fun writing the apps I'm developing specifically for this book.

0.4.4 I heard tkinter is ugly

It's true that plain tkinter is not going to win any beauty awards. It's old. The great thing is, tkinter now comes with a module called "ttk" which provides widgets which look native on Windows and OSX (tkinter itself looks very close to native on Linux already). Whilst this book doesn't cover ttk until the last project, after reading it you should be able to swap out the majority of widgets from earlier chapters to ttk's very easily. If you're following along on Windows or OSX don't be put off by the dated styling of tkinter's widgets; once you learn about using and styling ttk widgets in Chapter 7 you should grasp how to make tkinter look great on all platforms.

Chapter 1

Hello World

1.1 Basic Example

As is tradition with all programming books, we'll start with the classic Hello World example to introduce a few things. This will pop up a small window with "Hello World" written inside.

```
1 import tkinter as tk
2
3 root = tk.Tk()
4
5 label = tk.Label(root, text="Hello World", padx=10, pady=10)
6 label.pack()
7
8 root.mainloop()
```

Listing 1.1: Hello World

We start with `root=tk.Tk()` which creates the overall tk window. Then we define a `tk.Label()` which will hold our "Hello World" text. The first argument to a Tk widget is the parent in which it will be placed. In this case, we will just put it directly within the `root` instance. The `padx` and `pady` arguments add padding horizontally and vertically. `label.pack()` is then called as a way of placing the label into the root. Other ways of placing widgets, such as `grid()`, will be covered later. Finally `root.mainloop()` is responsible for showing the window.

Save and run this code and you should see a small window appear with "Hello World" inside, as show here:

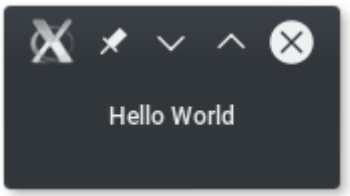

Figure 1.1: Our first Tk window

1.2 Using Classes

Whilst Tkinter code can be written using only functions, it's much better to use a class to keep track of all individual widgets which may need to reference each other. Without doing this, you need to rely on `global` or `nonlocal` variables, which gets ugly as your app grows. It also allows for much finer controls once your app gets more complex, allowing you to override default behaviours of Tkinter's own objects.

```
1  import tkinter as tk
2
3  class Root(tk.Tk):
4      def __init__(self):
5          super().__init__()
6
7          self.label = tk.Label(self, text="Hello World", padx=5, pady=5)
8
9          self.label.pack()
10
11 if __name__ == "__main__":
12     root = Root()
13     root.mainloop()
```

Listing 1.2: Hello World as a Class

The main code here is the same as above. The rest is simply creating a `Root` class inheriting from Tkinter's `Tk` and running its `mainloop` function as before. I've also included the standard `if "__name__" == __main__` line for familiarity.

The label now belongs to the Root, rather than being an independent variable. This allows us to reference it easily within methods of the Root class, such as an action we may bind to a `Button`, which could otherwise be out of scope if we were not using a class.

Running this code should produce the same small window as in the first example.

Now we've covered the very basics of making a window appear, let's dive in to something which can actually be used.

Chapter 2

A To-Do List

In this chapter we'll be creating a basic to-do list. Here we'll learn about the following:

- Allowing the user to enter text
- Binding functions to keypresses
- Dynamically generating widgets
- Scrolling an area
- Storing data (with sqlite)

2.1 A Basic List App

Your first app should look something like this:

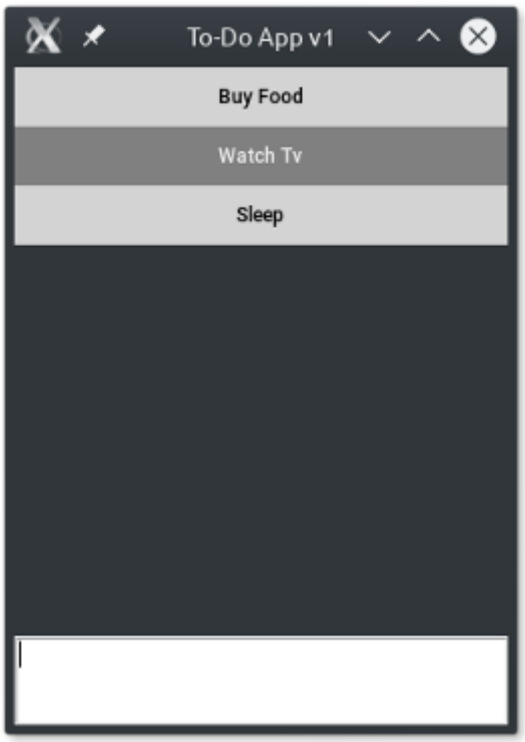

Figure 2.1: Our first To-Do App

Let's get right into the code for the first iteration.

```
import tkinter as tk

class Todo(tk.Tk):
    def __init__(self, tasks=None):
        super().__init__()

        if not tasks:
            self.tasks = []
        else:
            self.tasks = tasks

        self.title("To-Do App v1")
        self.geometry("300x400")

        todo1 = tk.Label(self, text="--- Add Items Here ---", bg="lightgrey", fg="black",
             pady=10)

        self.tasks.append(todo1)

        for task in self.tasks:
            task.pack(side=tk.TOP, fill=tk.X)

        self.task_create = tk.Text(self, height=3, bg="white", fg="black")

        self.task_create.pack(side=tk.BOTTOM, fill=tk.X)
        self.task_create.focus_set()

        self.bind("<Return>", self.add_task)

        self.colour_schemes = [{"bg": "lightgrey", "fg": "black"}, {"bg": "grey", "fg": "
            white"}]

    def add_task(self, event=None):
        task_text = self.task_create.get(1.0,tk.END).strip()

        if len(task_text) > 0:
            new_task = tk.Label(self, text=task_text, pady=10)

            _, task_style_choice = divmod(len(self.tasks), 2)

            my_scheme_choice = self.colour_schemes[task_style_choice]

            new_task.configure(bg=my_scheme_choice["bg"])
            new_task.configure(fg=my_scheme_choice["fg"])

            new_task.pack(side=tk.TOP, fill=tk.X)

            self.tasks.append(new_task)

        self.task_create.delete(1.0, tk.END)

if __name__ == "__main__":
    todo = Todo()
    todo.mainloop()
```

Listing 2.1: Our Initial To-Do Framework

2.1.1 __init__

We start off by defining our `Todo` class and initialising it with an empty list of tasks. If using a mutable data-type, such as a list, always ensure you set the default argument to `None` and convert it into a list within the `__init__` method, as unexpected behaviour can occur if you try and pass an empty list in. The reasons why are beyond the scope of this book, but you can find great explanations and examples online.

Next off, we set the title and size of the window. The app can be resized after creation if the user desires, so don't worry too much about getting the initial size perfect. The main reason for this is to signal to the user that the app should be vertically-oriented and prefers to be taller rather than wider.

A default task is added to our list to prevent it from just being a big blank space with a text box at the bottom, and to hint to the user what will happen when a task is added. We do this by creating a `Label`, adding it to our `tasks` list and packing it. The reason we use a loop to pack this item will become clear when we introduce persistent storage in a later section of this chapter. The `fg` (foreground) and `bg` (background) colours are set, and some vertical padding is added for aesthetics. The widgets are packed to the `TOP` of the window, and are set to fill in the `X` direction, i.e. horizontally, to ensure they are all of uniform width, and the background spans the entirety of the window.

The final widget we need is our `Text` box, which is what the user will type into. We shorten the default `height` to 3 to make it look a bit nicer, and specify the white background with black text to look more like traditional text inputs. After packing it at the `BOTTOM` of our window spanning the full `X` direction like our tasks, we call `focus_set` so that the cursor is inside the box when the window is opened. Without this, the user would have to click inside the box before they could type anything. We then bind the `Return` (or Enter) key to a function `add_item` which we will get to next. A note when binding - do not put the parentheses at the end of the function name. We want to pass the function itself across, but if we put the parentheses we will end up calling the function instead.

The last thing to do is define our colour schemes. This is used to better separate individual items from the list view. I've gone for light grey with black text, followed by darker grey with white text. Feel free to switch these up to suit your preferences. You may notice the default list item has the styling of the first scheme, so as to ensure it fits the pattern. The `colour_schemes` variable is a list of dictionaries containing a background and foreground colour, which we will use to alternate the styles when adding new tasks.

2.1.2 add_item

When adding a new item, the first thing to do is get the text which the user entered into our `Text` widget. The arguments here tell the widget how much of the text to grab. `1.0` tells it to begin at the first character, and the `END` constant tells it to look until the end of the box. We also call `strip()` on the result to remove the newline character which is entered when the user presses `Return` to submit the text, as well as any trailing space characters.

We need to check if the length of the entered text is greater than `0` to avoid letting the user add blank tasks. If this is true, then we create a new `Label` with the text entered by the user. The `divmod` function is used to determine whether we are on an even or odd number of total tasks, allowing us to set the correct styling to our new label. `Divmod` returns the quotient and remainder when the first argument is divided by the second. In our case, we want the remainder when the size of our list is divided by 2. The quotient is set to `_`, which is commonly used in python to denote a variable which we do not plan on using. The remainder is then used as the index of our `colour_schemes` list to grab the correct foreground and background colour dictionary. The `configure` method is used to set a property of a widget, just as you would pass the values in as keyword arguments when creating them initially. We set the foreground and background colours of our `Label` with the chosen dictionary's values, and then pack it the same way as our default item. Finally, we add this to the `tasks` variable so as to keep count of how many items we have.

We clear everything written in the `Text` widget outside of our `if` statement. This is to prevent the user from adding newlines before their task name by pressing `Return` before typing anything. We also want to clear it if they have entered a task, so they do not have to delete it manually before writing another.

2.1.3 Next Iteration

That's it for the first iteration of our to-do list! We now have a styled list of items which can be added to. Whilst playing with this example, you will probably notice that if you add too many items, you need to re-size the window to see any more. You also cannot delete any items which you may have completed. These will both be addressed next.

2.2 Scrolling and Deleting

A lot has changed from the previous iteration, so I will include the full code in this section. Your new To-do app can be written as follows:

```
1  import tkinter as tk
2  import tkinter.messagebox as msg
3
4  class Todo(tk.Tk):
5      def __init__(self, tasks=None):
6          super().__init__()
7
8          if not tasks:
9              self.tasks = []
10         else:
11             self.tasks = tasks
12
13         self.tasks_canvas = tk.Canvas(self)
14
15         self.tasks_frame = tk.Frame(self.tasks_canvas)
16         self.text_frame = tk.Frame(self)
17
18         self.scrollbar = tk.Scrollbar(self.tasks_canvas, orient="vertical", command=self.
               tasks_canvas.yview)
19
20         self.tasks_canvas.configure(yscrollcommand=self.scrollbar.set)
21
22         self.title("To-Do App v2")
23         self.geometry("300x400")
24
25         self.task_create = tk.Text(self.text_frame, height=3, bg="white", fg="black")
26
27         self.tasks_canvas.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
28         self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
29
30         self.canvas_frame = self.tasks_canvas.create_window((0, 0), window=self.
               tasks_frame, anchor="n")
31
32         self.task_create.pack(side=tk.BOTTOM, fill=tk.X)
33         self.text_frame.pack(side=tk.BOTTOM, fill=tk.X)
34         self.task_create.focus_set()
35
36         todo1 = tk.Label(self.tasks_frame, text="--- Add Items Here ---", bg="lightgrey",
                fg="black", pady=10)
37         todo1.bind("<Button-1>", self.remove_task)
38
39         self.tasks.append(todo1)
40
41         for task in self.tasks:
42             task.pack(side=tk.TOP, fill=tk.X)
43
44         self.bind("<Return>", self.add_task)
45         self.bind("<Configure>", self.on_frame_configure)
46         self.bind_all("<MouseWheel>", self.mouse_scroll)
47         self.bind_all("<Button-4>", self.mouse_scroll)
48         self.bind_all("<Button-5>", self.mouse_scroll)
49         self.tasks_canvas.bind("<Configure>", self.task_width)
50
51         self.colour_schemes = [{"bg": "lightgrey", "fg": "black"}, {"bg": "grey", "fg": "
               white"}]
52
53     def add_task(self, event=None):
54         task_text = self.task_create.get(1.0,tk.END).strip()
55
56         if len(task_text) > 0:
57             new_task = tk.Label(self.tasks_frame, text=task_text, pady=10)
```

```
58
59              self.set_task_colour(len(self.tasks), new_task)
60
61              new_task.bind("<Button-1>", self.remove_task)
62              new_task.pack(side=tk.TOP, fill=tk.X)
63
64              self.tasks.append(new_task)
65
66          self.task_create.delete(1.0, tk.END)
67
68      def remove_task(self, event):
69          task = event.widget
70          if msg.askyesno("Really Delete?", "Delete " + task.cget("text") + "?"):
71              self.tasks.remove(event.widget)
72              event.widget.destroy()
73              self.recolour_tasks()
74
75      def recolour_tasks(self):
76          for index, task in enumerate(self.tasks):
77              self.set_task_colour(index, task)
78
79      def set_task_colour(self, position, task):
80          _, task_style_choice = divmod(position, 2)
81
82          my_scheme_choice = self.colour_schemes[task_style_choice]
83
84          task.configure(bg=my_scheme_choice["bg"])
85          task.configure(fg=my_scheme_choice["fg"])
86
87      def on_frame_configure(self, event=None):
88          self.tasks_canvas.configure(scrollregion=self.tasks_canvas.bbox("all"))
89
90      def task_width(self, event):
91          canvas_width = event.width
92          self.tasks_canvas.itemconfig(self.canvas_frame, width = canvas_width)
93
94      def mouse_scroll(self, event):
95          if event.delta:
96              self.tasks_canvas.yview_scroll(int(-1*(event.delta/120)), "units")
97          else:
98              if event.num == 5:
99                  move = 1
100             else:
101                 move = -1
102
103             self.tasks_canvas.yview_scroll(move, "units")
104
105 if __name__ == "__main__":
106     todo = Todo()
107     todo.mainloop()
```

Listing 2.2: Our Scrolling To-Do

2.2.1 Canvases and Frames

With this re-write, I have introduced some new components - a `Canvas` and two `Frames`. A `Canvas` is a powerful general-use widget with many capabilities (usually graphical). We are using it here for its ability to scroll, which we need if we want to add a lot of apps to our list. A `Frame` is a layout component which can be used to group together multiple other widgets. As you will see in this case, we can actually use the `Canvas` to draw a `Frame` into our window, which is then able to bundle together all of our to-do items, allowing them to scroll independently of the `Text` widget we use to add new tasks.

2.2.2 `__init__`

As above, we now create a `Canvas` and two `Frames`, with one `Frame` parented to the canvas, and the other to the main window. We then make a `Scrollbar` object to allow scrolling of the page. We set the orientation and command to tell tkinter that we want a vertical scrollbar, scrolling in the y direction. We also configure our canvas to accept the `Scrollbar`'s values. We once again set the window title and size, and create our `Text` widget - this time parented to one of the frames (which will be packed to the bottom). Our `Canvas` is packed with instruction to fill all available space and expand as big as it can, and our `Scrollbar` follows, filling up the vertical space.

The next line looks a little strange. We use our `Canvas` to create a new window inside itself, which is our `Frame` holding the tasks. We create it at the coordinates `(0,0)` and anchor it to the top of the `Canvas` (the "n" here is for "north", so top-left would require "nw", and so on). One thing to note is that we do **not** pack our `tasks_frame`, as it will not appear, and we will be left scratching our heads as to where it is. This is something I learned the hard way!

After that, we pack our `Text` into its frame and then pack its frame to the `BOTTOM` of the window, with both filling the `X` direction. The default task is created and we bind the `self.remove_task` function to it being clicked (this will be covered below). We pack this, and then move on to a big block of binds. The `<MouseWheel>`, `<Button-4>` and `<Button-5>` binds handle scrolling, and the `<Configure>` binds handle keeping the `Canvas` as big as possible as the window changes size. The `<Configure>` event is fired when widgets change size (and on some platorms, location) and will provide the new width and height. The `<Return>` bind and `colour_schemes` remain from the previous example.

2.2.3 Handling Tasks

The `add_task` method is almost the same as the previous iteration, but the code for choosing the styling has been moved into a separate method - `set_task_colour` - so that it can be re-used after deleting tasks. Speaking of which, we have a `remove_task` method which will handle getting rid of the `Label` widget associated with the task. To avoid accidental removal, we use an `askyesno` pop-up message to double-check with the user that they wanted to delete that task (make sure you don't miss the new `import tkinter.messagebox as msg` statement at the top of the file). This will create a small notice with the title "Really Delete?" and the message "Delete <task>?" (where <task> will be the text within the `Label`) with the options "yes" and "no". Using the `if` statement around this means the indented code will only happen if the user presses "yes". Upon deletion, we recolour all remaining tasks in our alternating pattern, as otherwise the pattern would be broken by the removal.

2.2.4 Adjusting the canvas

Our `on_frame_configure` method is bound to our `root`'s `<Configure>` action, and will be called whenever the window is resized. It sets the scrollable region for our canvas, and uses the `bbox` (bounding box) to specify that we want the entire canvas to be scrollable. The `task_width` method is bound to the `Canvas`'s `<Configure>`, and is responsible for ensuring the task `Labels` stay at the full width of the canvas, even after stretching the window.

2.2.5 Mouse scrolling

Our final method, `mouse_scroll`, is how we bind scrolling to the mouse wheel as well as the scrollbar. This is bound to `<MouseWheel>` for Windows and OSX, and to `<Button-4>` and `<Button-5>` for Linux. We then simply call the `Canvas`' `yview_scroll` method based upon whether we receive a `delta` or a `num` within the event. Here on Linux I get a `num`. The delta is usually 120 or -120, so is divided by 120 for more precise scrolling, and multiplied by -1 to adjust the direction.

2.2.6 Next Iteration

Our final iteration will handle saving and retrieving values from a `sqlite` database. I have left this until last because it's not strictly `tkinter` related, and so you are free to skip this section if you have no interest in learning about databases, or you already know enough to figure out how to do this on your own. If you think the latter is true, please do go ahead and try as an exercise before reading this section.

2.3 Permanent Storage

There are only a few small changes to our existing methods in this iteration, so I will not re-print the whole class. If you wish to follow along, start with your code from the previous version, make the changes listed in this section, and add any other new methods to the end of our `Todo` class. As a reminder, the full code will be available on Github at http://github.com/Dvlv/Tkinter-By-Example as `Chapter2-3.py`.

```
import tkinter as tk
import tkinter.messagebox as msg
import os
import sqlite3

class Todo(tk.Tk):
    def __init__(self, tasks=None):
        ...

        self.title("To-Do App v3")

        ...

        self.colour_schemes = [{"bg": "lightgrey", "fg": "black"}, {"bg": "grey", "fg": "
            white"}]

        current_tasks = self.load_tasks()
        for task in current_tasks:
            task_text = task[0]
            self.add_task(None, task_text, True)

        ...

    def add_task(self, event=None, task_text=None, from_db=False):
        if not task_text:
            task_text = self.task_create.get(1.0,tk.END).strip()

        if len(task_text) > 0:
            new_task = tk.Label(self.tasks_frame, text=task_text, pady=10)

            self.set_task_colour(len(self.tasks), new_task)

            new_task.bind("<Button-1>", self.remove_task)
            new_task.pack(side=tk.TOP, fill=tk.X)

            self.tasks.append(new_task)

            if not from_db:
                self.save_task(task_text)

        self.task_create.delete(1.0, tk.END)

    def remove_task(self, event):
        task = event.widget
        if msg.askyesno("Really Delete?", "Delete " + task.cget("text") + "?"):
            self.tasks.remove(event.widget)

            delete_task_query = "DELETE FROM tasks WHERE task=?"
            delete_task_data = (task.cget("text"),)
            self.runQuery(delete_task_query, delete_task_data)

            event.widget.destroy()

            self.recolour_tasks()

    ...

    def save_task(self, task):
        insert_task_query = "INSERT INTO tasks VALUES (?)"
```

```
59          insert_task_data = (task,)
60          self.runQuery(insert_task_query, insert_task_data)
61
62      def load_tasks(self):
63          load_tasks_query = "SELECT task FROM tasks"
64          my_tasks = self.runQuery(load_tasks_query, receive=True)
65
66          return my_tasks
67
68      @staticmethod
69      def runQuery(sql, data=None, receive=False):
70          conn = sqlite3.connect("tasks.db")
71          cursor = conn.cursor()
72          if data:
73              cursor.execute(sql, data)
74          else:
75              cursor.execute(sql)
76
77          if receive:
78              return cursor.fetchall()
79          else:
80              conn.commit()
81
82          conn.close()
83
84      @staticmethod
85      def firstTimeDB():
86          create_tables = "CREATE TABLE tasks (task TEXT)"
87          Todo.runQuery(create_tables)
88
89          default_task_query = "INSERT INTO tasks VALUES (?)"
90          default_task_data = ("--- Add Items Here ---",)
91          Todo.runQuery(default_task_query, default_task_data)
92
93
94  if __name__ == "__main__":
95      if not os.path.isfile("tasks.db"):
96          Todo.firstTimeDB()
97      todo = Todo()
98      todo.mainloop()
```

Listing 2.3: Database Integration

2.3.1 runQuery

Let's start by explaining the database handling. Our `runQuery` method is a fairy generic database handling method. It takes an sql string, some data to format into the sql string, and `receive` which indicates to the method whether or not it needs to return any data (from a `SELECT` statement). We first connect to our database file, in this case `tasks.db`, and receive a `cursor`. The `cursor` is used to execute queries against the database and sometimes return data. We then close off our connection at the end to reduce resource usage. This is a static method so that it can be called by our proceeding `firstTimeDb` method, which needs to be called before our `__init__`, and so is also static.

2.3.2 firstTimeDb

This function is used to create the database file, `tasks.db`, if it does not already exist. We also put our old default task, `--- Add Tasks Here ---`, in this method so that it appears when the user first loads the app, but is permanently deletable like other tasks.

2.3.3 __init__

We start by just updating the window's title bar to the 3rd version. We move the existing `colour_schemes` variable to above the new code which will populate our existing tasks, so that we can use it during the initial set-up. Without doing this, we would get an error when we reference it via `add_task`. Instead of the hard-coded default task, we now fetch existing tasks from the database with `load_tasks`, then iterate through them, passing each to our slightly altered `add_task` method.

2.3.4 add_task and remove_task

To prevent re-writing most of this code in our `__init__` method, we have added two new parameters to `add_task`: `task_text` and `from_db`. This allows us to pass in text independent of our `Text` widget, and to prevent re-saving tasks to the database which originated from there. Before destroying our widget inside `remove_task`, we grab its text and remove it from the database too.

2.3.5 save_task and load_tasks

These two methods deal with database access. `save_task` will add a new task into our database, and `load_tasks` is called in our `__init__` method to retrieve all saved tasks when loading the app. These two methods ensure that the task list displays the same when the user closes then re-opens the app.

2.3.6 The final app

That's it for our to-do list. We now have a to-do application which can save and retrieve tasks which remain after closing the app. We have learned how to layout multiple widgets with `Frames` and the `pack` method, how to make a scrollable area which maintains its size when the window is resized, how to bind methods to user inputs and `tkinter`'s own events, and how to dynamically add and remove widgets based on user actions. If you read the final section, you will also know how to integrate `tkinter` nicely with a `sqlite` database. Next up we will create an app which utilises a tabbed interface, also known as a `Notebook`.

2.3.7 Further Development

If you'd like to continue work on this project as an exercise, try the following:

- Prevent duplicate tasks by using a database look-up before adding a new task.
- Give each task a "Delete" button instead of the on-click event on the Label itself (Buttons will be covered next chapter).
- Instead of destroying tasks, mark them as "finished" using a column in the database and display them as "greyed out".

- Add a "category" for each task and colour the task based on the category instead of using the pattern (maybe separate them with a border).

Chapter 3

A Multi-Language Translation Tool

In this chapter we'll be creating a tool which will translate english text into multiple other languages using the Google Translate API. Here we'll learn about the following:

- Creating a tabbed interface
- Creating a Menu
- Creating a pop-up window
- Accessing the Clipboard
- Calling APIs with `requests`

3.1 A Single-Translation Interface

We'll start with a simple app which translates to one language (italian). Your first app should look something like this:

Figure 3.1: A two-tabbed translator (English)

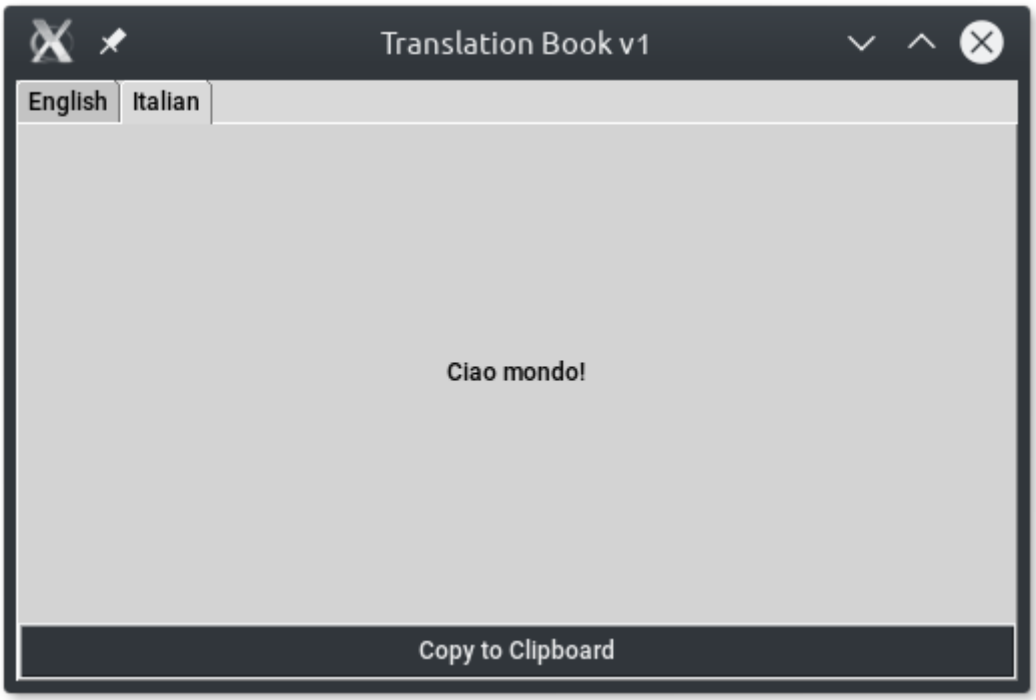

Figure 3.2: A two-tabbed translator (Italian)

```
1  import tkinter as tk
2  from tkinter import messagebox as msg
3  from tkinter.ttk import Notebook
4
5  import requests
6
7  class TranslateBook(tk.Tk):
8      def __init__(self):
9          super().__init__()
10
11         self.title("Translation Book v1")
12         self.geometry("500x300")
13
14         self.notebook = Notebook(self)
15
16         english_tab = tk.Frame(self.notebook)
17         italian_tab = tk.Frame(self.notebook)
18
19         self.translate_button = tk.Button(english_tab, text="Translate", command=self.
               translate)
20         self.translate_button.pack(side=tk.BOTTOM, fill=tk.X)
21
22         self.english_entry = tk.Text(english_tab, bg="white", fg="black")
23         self.english_entry.pack(side=tk.TOP, expand=1)
24
25         self.italian_copy_button = tk.Button(italian_tab, text="Copy to Clipboard",
               command=self.copy_to_clipboard)
26         self.italian_copy_button.pack(side=tk.BOTTOM, fill=tk.X)
27
28         self.italian_translation = tk.StringVar(italian_tab)
29         self.italian_translation.set("")
30
31         self.italian_label = tk.Label(italian_tab, textvar=self.italian_translation, bg="
               lightgrey", fg="black")
32         self.italian_label.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
33
34         self.notebook.add(english_tab, text="English")
```

```
35          self.notebook.add(italian_tab, text="Italian")
36
37          self.notebook.pack(fill=tk.BOTH, expand=1)
38
39      def translate(self, target_language="it", text=None):
40          if not text:
41              text = self.english_entry.get(1.0, tk.END)
42
43          url = "https://translate.googleapis.com/translate_a/single?client=gtx&sl={}&tl
                =={}&dt=t&q={}".format("en", target_language, text)
44
45          try:
46              r = requests.get(url)
47              r.raise_for_status()
48              translation = r.json()[0][0][0]
49              self.italian_translation.set(translation)
50              msg.showinfo("Translation Successful", "Text successfully translated")
51          except Exception as e:
52              msg.showerror("Translation Failed", str(e))
53
54      def copy_to_clipboard(self, text=None):
55          if not text:
56              text = self.italian_translation.get()
57
58          self.clipboard_clear()
59          self.clipboard_append(text)
60          msg.showinfo("Copied Successfully", "Text copied to clipboard")
61
62
63  if __name__ == "__main__":
64      translatebook = TranslateBook()
65      translatebook.mainloop()
```

Listing 3.1: Our first translation app

3.1.1 requests

We now import and use the `requests` module. If you do not have this installed, you can get it with pip (`pip install requests`).

3.1.2 __init__

Hopefully most the `__init__` should look familiar to you by now. The first new bit is the creation of a `Notebook`, which is what holds our tabs. The contents of each notebook tab is simply a `Frame`, each of which holds two elements. Our `english_frame` holds a `Text` widget, allowing the user to enter some text, and a `Button` which triggers the translation. The `command` argument supplied to a `Button` is the function which we want to be called when it is clicked. An important thing to remember is to **not** put the parentheses at the end of the function name, as this will actually call the function and bind the result (we want to bind the function itself). This is the same potential mistake as when binding with the `bind` method from chapter 1.

Our `italian_frame` holds an expanded `Label` instead of a `Text` input, as we don't want to be able to alter the translated text, as well as a `Button` which will copy the translated text to our computer's clipboard.

Another new thing here is the use of a `StringVar`. As you may be able to guess from the name, this is like a sophisticated container for a string variable, which allows us the change the text of a `Label` without needing to re-configure it. Its other great use is changing the text of multiple `Labels` (which need to say the same thing) all at once, and we can also fire callbacks whenever the variable changes. In our case, the `StringVar` is used to update the `Label` containing our italian translation, and to grab the text back out to put onto our clipboard (as we'll see later).

Instead of packing our two frames, we just add them to our notebook and pass the `text` (i.e. the name of the tab) along with them, before finally packing our `Notebook`. Hopefully you should have a good idea of the use-cases of this app just from the `__init__` method.

3.1.3 translate

Much like with our to-do app, we grab the user's text from our `Text` widget, but we won't clear it this time in case they've typed something really long and want to add something after translation. We next create the URL to access google translate's API with the `format` method, passing in our original language code ("en"), target language code (defaults to "it", but we will specify this when adding another tab next iteration) and our text to be translated (which we grabbed from the `Text` widget earlier). We visit this URL using the `requests` module's `get` method. The `raise_for_status` method will raise an `Exception` should we recieve an error when calling the API, such as a 404 if there's a typo. For this reason, we've put our code in a `try / except` block so that we can gracefully alert the user via a `messagebox` if there's a problem. If no `Exceptions` are raised, we use the `json` method of `requests` to parse the json-formatted response from the API into a nice block of python lists. The translation is in the first element of the first element of the first list (not too graceful, I know!), hence the chaining of `[0][0][0]`. If you wish to look at the response, add a `print(translation)` on the next line. We finish up by setting the translated text as the value of our `StringVar` and showing the user a success message so that they know the other tabs have updated.

3.1.4 copy_to_clipboard

This is the function bound to the `Button` in our italian tab. We simply grab the `StringVar`'s value (which our `Label` holds) and use `tkinter` to add the text into our computer's clipboard. I originally intended to use the `pyperclip` module to handle the clipboard, but then I found out that `tkinter` can handle it already - super handy!

3.1.5 Next Iteration

Now that we have a proof-of-concept for our translator, we'll go deeper in and set up a second language for us to translate to, as well as a menu for us to pick languages from.

3.2 Three Tabs and a Menu

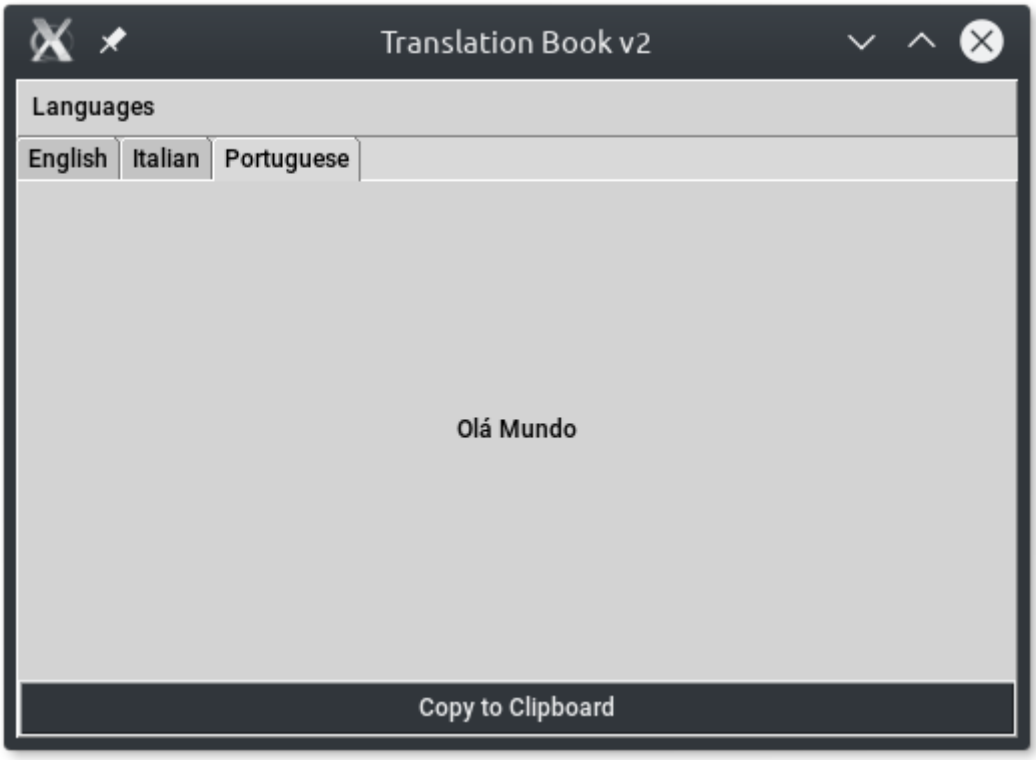

Figure 3.3: A portuguese translation in our notebook

Our next iteration boasts a menu bar at the top, and the ability to translate to both italian and portuguese at once. After running this iteration, click the "Languages" menu - you should see a "Portuguese" option. Selecting this will add a third tab to our `Notebook`. If we follow the same translation process as before, we will now see both the italian and portuguese tabs are updated with the translations. Neat. Whilst this is not yet fully dynamic, we've laid out some groundwork for alternative translations. Let's take a look at the code changes which make this possible.

```
...

class TranslateBook(tk.Tk):
    def __init__(self):

        ...

        self.menu = tk.Menu(self, bg="lightgrey", fg="black")

        self.languages_menu = tk.Menu(self.menu, tearoff=0, bg="lightgrey", fg="black")
        self.languages_menu.add_command(label="Portuguese", command=self.
            add_portuguese_tab)

        self.menu.add_cascade(label="Languages", menu=self.languages_menu)

        self.config(menu=self.menu)

        ...

        self.italian_translation = tk.StringVar(italian_tab)
        self.italian_translation.set("")

        self.translate_button = tk.Button(english_tab, text="Translate", command=lambda
            langs=["it"], elems=[self.italian_translation]: self.translate(langs, None,
            elems))

        ...

    def translate(self, target_languages=None, text=None, elements=None):
        if not text:
            text = self.english_entry.get(1.0, tk.END).strip()
        if not elements:
            elements = [self.italian_translation]
        if not target_languages:
            target_languages = ["it"]

        url = "https://translate.googleapis.com/translate_a/single?client=gtx&sl={}&tl
            ={}&dt=t&q={}"

        try:
            for code, element in zip(target_languages, elements):
                full_url = url.format("en", code, text)
                r = requests.get(full_url)
                r.raise_for_status()
                translation = r.json()[0][0][0]
                element.set(translation)
        except Exception as e:
            msg.showerror("Translation Failed", str(e))
        else:
            msg.showinfo("Translations Successful", "Text successfully translated")

    def copy_to_clipboard(self, text=None):
        ...

    def add_portuguese_tab(self):
        portuguese_tab = tk.Frame(self.notebook)
        self.portuguese_translation = tk.StringVar(portuguese_tab)
        self.portuguese_translation.set("")

        self.portuguese_copy_button = tk.Button(portuguese_tab, text="Copy to Clipboard",
             command=lambda: self.copy_to_clipboard(self.portuguese_translation.get()))
        self.portuguese_copy_button.pack(side=tk.BOTTOM, fill=tk.X)

        self.portuguese_label = tk.Label(portuguese_tab, textvar=self.
            portuguese_translation, bg="lightgrey", fg="black")
        self.portuguese_label.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
```

```
61
62          self.notebook.add(portuguese_tab, text="Portuguese")
63
64          self.languages_menu.entryconfig("Portuguese", state="disabled")
65
66          self.translate_button.config(command=lambda langs=["it","pt"], elems=[self.
                italian_translation, self.portuguese_translation]: self.translate(langs, None
                , elems))
67
68
69  if __name__ == "__main__":
70      translatebook = TranslateBook()
71      translatebook.mainloop()
```

Listing 3.2: Our Translator with a Menu

3.2.1 __init__

We now encounter a new `tkinter` widget - a `Menu`. A `Menu` is essentially a container for a list of buttons. We start by declaring our "overall" menu, `self.menu`, which will hold our submenu, `self.languages_menu`. We set `tearoff` to 0 so that the user can't drag-and-drop the languages submenu out of the main menu. We then add a command (essentially a button) called Portuguese. We bind the `add_portuguese_tab` method to this button, again making sure not to call the function. We then use `add_cascade` to place our submenu into our main bar. We finish up by calling `self.configure(menu=self.menu)` to set the root window's menu to our overall menu.

The only other change to this method is the moving of the `italian_translation StringVar` to above our `translate_button` so that we can use it in the `command`. Speaking of which, we've now changed this to a `lambda` which calls the new-and-improved translate method with a couple of lists as arguments. Let's look into `translate` now.

3.2.2 translate

Our `translate` now takes another argument - `elements` - which is a list of `StringVars` to update with a translation. The `target_languages` argument is now expected to be a list of language codes, and the name has been pluralised to reflect this.

Our `url` is no longer formatted upon creation, but is instead left with the placeholders in. We use `zip` to combine our lists of language codes and `StringVar` elements into the correct pairs and then use them to format our URL, parse out the translation, and update the `StringVar` as before - but this time in a loop, allowing us to do this for any number of languages. You may not have come across an `else` by a `try / except` block before. The purpose of the `else` is to execute code only if there was no exception caught in the `except`. We've put our success `messagebox` in this `else` because we only want it to show once, so it couldn't be left inside the `for` loop, and we don't want it to show if, say, the first translation worked but the second did not. Out there in the `else` it should not be able to mislead the user into thinking the translation was successful if it wasn't, and will only appear once at the end of the process.

3.2.3 add_portuguese_tab

This is the function called when we choose our "Portuguese" option from our "Languages" menu. A lot of the code here looks just like the italian code from our `__init__`. Since our `copy_to_clipboard` method still has all of the defaults set to the italian translations, our `portuguese_copy_button` instead uses a lambda to call it with the `text` argument as the value of its `portuguese_translation StringVar`.

At the end of the function we disable the "Portuguese" entry in our "Languages" menu. Without this we could create multiple Portuguese tabs, which is pointless. We finish off by changing the `command` of

our translate button to a new `lambda` which contains both the italian and portuguese language codes and `StringVars`.

3.2.4 Next Iteration

You may notice this code feels a bit hacky. The `add_portuguese_tab` function knows (well, assumes) that we have an italian tab, and directly modifies our translate button too. In order to generalise this for re-use we're going to look at making each translation `Frame` its own class - allowing us to make any language supported by google translate and add it as a tab to our notebook. The reason we didn't do this all in one go was so that we could meet the `Menu` widget and lay the groundwork for dynamically adding tabs before a big overhaul of the app.

3.3 A Truly Dynamic App

Our code is now split into 3 classes which I will cover separately. The executable code for this section is all in Chapter3-3.py for those downloading it from Github. It is best practice to keep to one class per file, but for the sake of book simplicity I've combined them. We'll start this section off by looking at the new LanguageTab class.

3.3.1 The LanguageTab

```
1  class LanguageTab(tk.Frame):
2      def __init__(self, master, lang_name, lang_code):
3          super().__init__(master)
4
5          self.lang_name = lang_name
6          self.lang_code = lang_code
7
8          self.translation_var = tk.StringVar(self)
9          self.translation_var.set("")
10
11         self.translated_label = tk.Label(self, textvar=self.translation_var, bg="
               lightgrey", fg="black")
12
13         self.copy_button = tk.Button(self, text="Copy to Clipboard", command=self.
               copy_to_clipboard)
14
15         self.copy_button.pack(side=tk.BOTTOM, fill=tk.X)
16         self.translated_label.pack(side=tk.TOP, fill=tk.BOTH, expand=1)
17
18     def copy_to_clipboard(self):
19         root = self.winfo_toplevel()
20         root.clipboard_clear()
21         root.clipboard_append(self.translation_var.get())
22         msg.showinfo("Copied Successfully", "Text copied to clipboard")
```

Listing 3.3: An Independent Language Tab

Our `LanguageTab` class is built on top of a `Frame`, since that's what we add into our `Notebook`. It holds a reference to the full name of the language (for the tab name) and its short code for the google translate API. It is responsible for its own `StringVar`, `Label` and `Button`, as well as the command bound to the `Button`

The `copy_to_clipboard` method needs to access the root window, i.e. our `TranslateBook` instance, because that's what has control over the clipboard. We grab this with the `winfo_toplevel` method, then use the same code as before to put our `StringVar`'s contents onto the clipboard.

Now we'll jump back to the main `TranslateBook` class which handles our root window.

3.3.2 The TranslateBook

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

        self.title("Translation Book v3")
        self.geometry("500x300")

        self.menu = tk.Menu(self, bg="lightgrey", fg="black")

        self.languages_menu = tk.Menu(self.menu, tearoff=0, bg="lightgrey", fg="black")
        self.languages_menu.add_command(label="Add New", command=self.
            show_new_language_popup)
        self.languages_menu.add_command(label="Portuguese", command=lambda: self.
            add_new_tab(LanguageTab(self, "Portuguese", "pt")))

        self.menu.add_cascade(label="Languages", menu=self.languages_menu)

        self.config(menu=self.menu)

        self.notebook = Notebook(self)

        self.language_tabs = []

        english_tab = tk.Frame(self.notebook)

        self.translate_button = tk.Button(english_tab, text="Translate", command=self.
            translate)
        self.translate_button.pack(side=tk.BOTTOM, fill=tk.X)

        self.english_entry = tk.Text(english_tab, bg="white", fg="black")
        self.english_entry.pack(side=tk.TOP, expand=1)

        self.notebook.add(english_tab, text="English")

        self.notebook.pack(fill=tk.BOTH, expand=1)

    def translate(self, text=None):
        if len(self.language_tabs) < 1:
            msg.showerror("No Languages", "No languages added. Please add some from the
                menu")
            return

        if not text:
            text = self.english_entry.get(1.0, tk.END).strip()

        url = "https://translate.googleapis.com/translate_a/single?client=gtx&sl={}&tl
            ={}&dt=t&q={}"

        try:
            for language in self.language_tabs:
                full_url = url.format("en", language.lang_code, text)
                r = requests.get(full_url)
                r.raise_for_status()
                translation = r.json()[0][0][0]
                language.translation_var.set(translation)
        except Exception as e:
            msg.showerror("Translation Failed", str(e))
        else:
            msg.showinfo("Translations Successful", "Text successfully translated")

    def add_new_tab(self, tab):
        self.language_tabs.append(tab)
        self.notebook.add(tab, text=tab.lang_name)

```

```
60          try:
61              self.languages_menu.entryconfig(tab.lang_name, state="disabled")
62          except:
63              # language isn't in menu.
64              pass
65
66      def show_new_language_popup(self):
67          NewLanguageForm(self)
```

Listing 3.4: Our Main Class

__init__

We've added a new item to our `languages_menu` - add new - which will be covered with our final class `NewLanguageForm`. We've also re-written our portuguese entry to use a new method `add_new_tab`. We no longer make everything for our italian tab since this is handled with the `LanguageTab` class, we instead keep a list of tabs inside `self.language_tabs`. Since our english tab is different, we still have all of the set up of that here.

translate

This should still look very familiar. Instead of passing in a list of language codes and elements, we just grab our list of `language_tabs` and pull the codes and elements from each instance. If we have no language tabs a `messagebox` will alert the user to add one first and exit the method with `return`.

add_new_tab

We pass this method a `LanguageTab` object and it gets appended to our `language_tabs` list and added to our `Notebook`. We also try to disable the `menu` entry if it exists. We don't mind if this fails, as it likely means the language was created outside of the menu and there's no entry to disable, so we can just `pass` if an `Exception` is thrown.

show_new_language_popup

All we need to do here is create the `NewLanguageForm` instance which will handle everything else. Let's look at this now.

3.3.3 NewLanguageForm

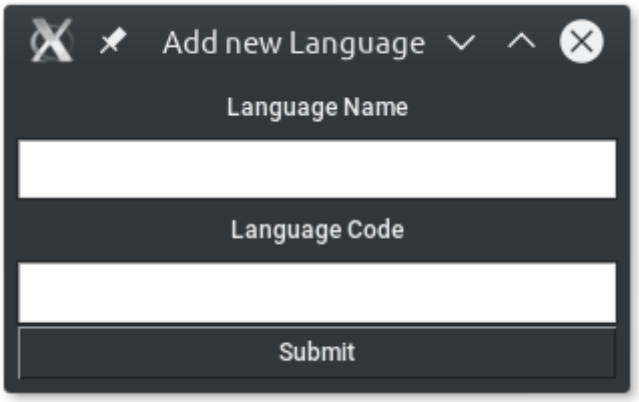

Figure 3.4: Our Add New Language Form

```
class NewLanguageForm(tk.Toplevel):
    def __init__(self, master):
        super().__init__()

        self.master = master

        self.title("Add new Language")
        self.geometry("300x150")

        self.name_label = tk.Label(self, text="Language Name")
        self.name_entry = tk.Entry(self, bg="white", fg="black")
        self.code_label = tk.Label(self, text="Language Code")
        self.code_entry = tk.Entry(self, bg="white", fg="black")
        self.submit_button = tk.Button(self, text="Submit", command=self.submit)

        self.name_label.pack(fill=tk.BOTH, expand=1)
        self.name_entry.pack(fill=tk.BOTH, expand=1)
        self.code_label.pack(fill=tk.BOTH, expand=1)
        self.code_entry.pack(fill=tk.BOTH, expand=1)
        self.submit_button.pack(fill=tk.X)

    def submit(self):
        lang_name = self.name_entry.get()
        lang_code = self.code_entry.get()

        if lang_name and lang_code:
            new_tab = LanguageTab(self.master, lang_name, lang_code)
            self.master.languages_menu.add_command(label=lang_name, command=lambda: self.
                master.add_new_tab(new_tab))
            msg.showinfo("Language Option Added", "Language option " + lang_name + "
                added to menu")
            self.destroy()
        else:
            msg.showerror("Missing Information", "Please add both a name and code")
```

Listing 3.5: Our Translator with a Menu

As you should be able to interpret from the code, we have a small window with 2 `Labels`, 2 `Entries` and a `Button`. An `Entry` is just a `Text` widget which is only one line. If you're familiar with HTML, think of an `Entry` as an `input[type="text"]` and a `Text` as a `textarea`. Our `__init__` just sets our window title and size, creates the widgets, and packs them all. The `master` argument to here is our `TranslateBook` instance, as the `submit` method needs to access its `languages_menu`

Our `submit` method is called by our `Button`. It grabs the text from our two `Entries` and creates a

LanguageTab instance from them. It then accesses our TranslateBook's languages_menu and adds the newly created LanguageTab instance as an option. Finally it shows a success messagebox and destroys itself (so the user doesn't have to close it manually). If you don't like this, you could always clear the Entries and leave the window open for the user to add another language straight after. If the user hasn't filled out one of the Entries a messagebox will let them know that they are both needed.

3.3.4 Running this version

In our old if __name__ == "__main__" statement we just created a TranslateBook instance and called its mainloop. If we want tabs to appear by default, like our italian tab originally, we need to create a LanguageTab instance and then use add_new_tab to add it to our TranslateBook before calling mainloop. In Chapter3-3.py you will see I have done this with the italian tab as before.

If you don't know of a language and code to test the NewLanguageForm out with, try "Spanish" and "es". Keep in mind that we only add the new language as a menu option, so it will not appear in your Notebook straight away, you must pick it from the menu first.

3.3.5 Further Development

If you'd like to continue work on this project as an exercise, try the following:

- Import ttk and adjust the app to use ttk's widgets (you will see a small attempt at this with Chapter3-3-ttk.py on Github, as I eventually deemed it unworthy of its own section).
- Bind the relevant Button functionality to the Return key.
- Before adding a new language validate that the short code added exists for the google translate api.
- Remember the app's previous state with sqlite (i.e. which tabs were added and which languages were available in the menu).
- Add a "Remove a Language" Menu which lists the enabled languages and lets the user remove one.

Chapter 4

A Point-and-Click Game

In this chapter we'll be creating one of those point-and-click puzzle games. Here we'll learn about the following:

- Handling images
- Drawing on and updating a Canvas

4.1 The Initial Concept

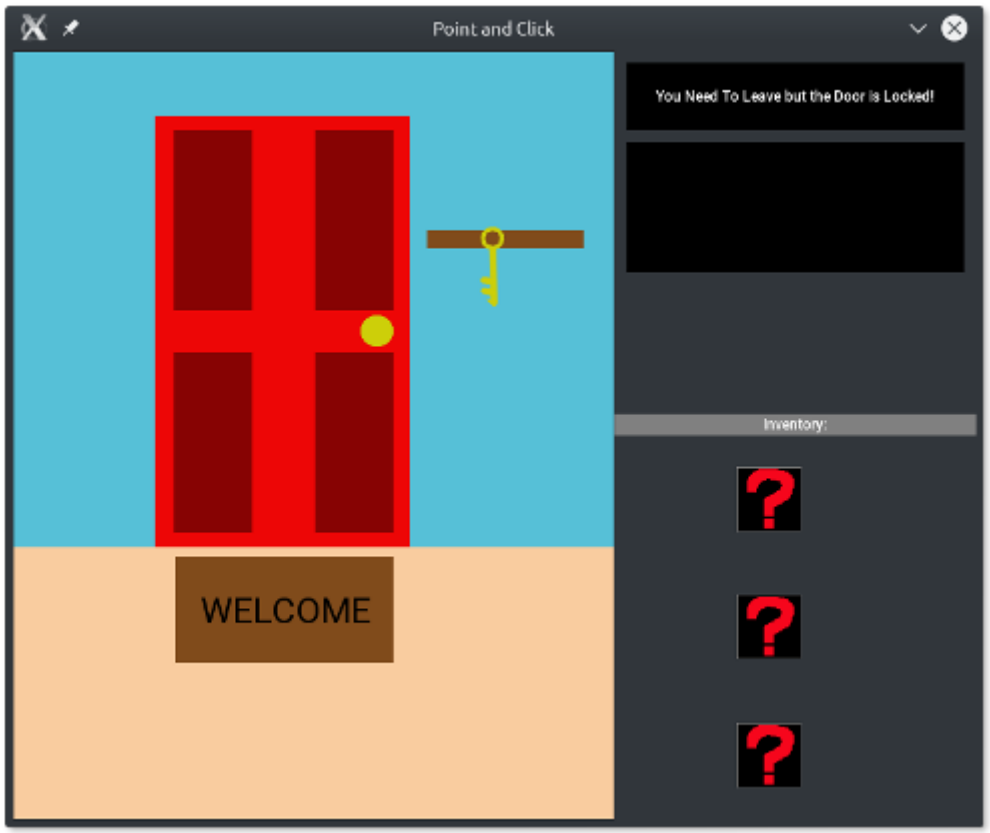

Figure 4.1: Our Point-and-Click Game

The concept I chose for our game is like a super simple version of the "escape the room" puzzle games. You see a door and need to escape. If you are following along, give the game a try before you begin. It's Chapter4-1.py from Github. When writing out this code, feel free to use my (amazing) artwork if you don't fancy drawing anything yourself.

```
import tkinter as tk
from tkinter import font

class GameScreen():
    def __init__(self, master, image, roi, inventory_item=None, help_text=None):
        self.master = master
        self.roi = roi
        self.image = tk.PhotoImage(file=image)
        self.inventory_item = inventory_item
        self.help_text = help_text

    def on_click(self, event):
        if (self.roi[0] <= event.x <= self.roi[2]
            and self.roi[1] <= event.y <= self.roi[3]):

            if self.inventory_item:
                self.master.add_inventory_item(self.inventory_item)
            self.master.show_next_screen()


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

        self.inventory_slots = []
        self.inventory_slots_in_use = []
        self.current_screen_number = 0
        self.success_font = font.Font(family="ubuntu", size=50, weight=font.BOLD)

        self.title("Point and Click")
        self.geometry("800x640")
        self.resizable(False, False)

        self.key_image = tk.PhotoImage(file="assets/key.png")
        self.question_mark_image = tk.PhotoImage(file="assets/questionmark.png")

        self.screen = tk.Canvas(self, bg="white", width=500, height=800)
        self.right_frame = tk.Frame(self, width=300, height=800)
        self.right_frame.pack_propagate(0)

        self.help_var = tk.StringVar(self.right_frame)
        self.help_var.set("Try Clicking Something")

        self.help_box = tk.Label(self.right_frame, textvar=self.help_var, background="
            black", foreground="white", padx=10, pady=20)
        self.help_box.pack(side=tk.TOP, fill=tk.X, padx=10, pady=10)

        inventory_title = tk.Label(self.right_frame, text="Inventory:", background="grey"
            , foreground="white")

        inventory_space = tk.Frame(self.right_frame, background="lightgrey", width=300,
            height=320)
        inventory_space.pack_propagate(0)

        inventory_space.pack(side=tk.BOTTOM)
        inventory_title.pack(side=tk.BOTTOM, fill=tk.X)

        inventory_slot_1 = tk.Button(inventory_space, image=self.question_mark_image,
            width=50, height=50)
        inventory_slot_2 = tk.Button(inventory_space, image=self.question_mark_image,
            width=50, height=50)
        inventory_slot_3 = tk.Button(inventory_space, image=self.question_mark_image,
            width=50, height=50)

        inventory_slot_1.pack(pady=(40,20), padx=20)
        inventory_slot_2.pack(pady=20, padx=20)
```

```
61          inventory_slot_3.pack(pady=(20,0), padx=20)
62
63          self.inventory_slots.append(inventory_slot_1)
64          self.inventory_slots.append(inventory_slot_2)
65          self.inventory_slots.append(inventory_slot_3)
66
67          self.right_frame.pack(side=tk.RIGHT)
68          self.screen.pack(side=tk.LEFT)
69
70          self.screen.bind("<Button-1>", self.handle_click)
71
72      def handle_click(self, event):
73          self.active_screen.on_click(event)
74
75      def set_game_screens(self, game_screens):
76          self.game_screens = game_screens
77
78      def display_screen(self, game_screen_number):
79          self.active_screen = self.game_screens[game_screen_number]
80          self.screen.delete("all")
81          self.screen.create_image((250,400), image=self.active_screen.image)
82          self.help_var.set(self.active_screen.help_text)
83
84      def show_next_screen(self):
85          self.current_screen_number += 1;
86          if self.current_screen_number < len(self.game_screens):
87              self.display_screen(self.current_screen_number)
88          else:
89              self.screen.delete("all")
90              self.screen.configure(bg="black")
91              self.screen.create_text((250,300), text="You Win!", font=self.success_font,
                    fill="white")
92
93      def add_inventory_item(self, item_name):
94          next_available_inventory_slot = len(self.inventory_slots_in_use)
95          if next_available_inventory_slot < len(self.inventory_slots):
96              next_slot = self.inventory_slots[next_available_inventory_slot]
97
98              if item_name == "key":
99                  next_slot.configure(image=self.key_image)
100
101             self.inventory_slots_in_use.append(item_name)
102
103     def play(self):
104         if not self.game_screens:
105             print("No screens added!")
106         else:
107             self.display_screen(0)
108
109
110 if __name__ == "__main__":
111     game = Game()
112
113     scene1 = GameScreen(game, "assets/scene1.png", (378,135,427,217), "key", "You Need To
            Leave but the Door is Locked!")
114     scene2 = GameScreen(game, "assets/scene2.png", (117,54,329,412), None, "You Got the
        Key!")
115     scene3 = GameScreen(game, "assets/scene3.png", (117,54,329,412), None, "The Door is
        Open!")
116
117     all_screens = [scene1, scene2, scene3]
118
119     game.set_game_screens(all_screens)
120     game.play()
```

```
121     game.mainloop()
```

Listing 4.1: Our Game

4.1.1 GameScreen

The GameScreen Class is essentially a nice container around the attributes associated with each screen. It holds a reference to our main Game object, the image to display for this screen (I'll cover PhotoImages next), the region-of-interest (i.e. where to click in order to advance), an item to be picked up, and the help text to display. The on_click function is sent the click event from the Game's Canvas. It compares the coordinates of the clicked point of the Canvas to its region-of-interest, then advances the game if the correct area was clicked. If the screen holds an inventory item it is added to the Game's inventory before advancing. I debated with myself whether or not to handle this logic within the Game itself, but have decided it looks a bit neater here.

4.1.2 Game

Our Game object defines the main window and layout, as well as handles tracking and progressing in the game. Let's break it down a bit:

__init__

We begin with creating some empty lists for our inventory and used-inventory (more on this later). We initialise the current screen to 0 and create a Font which will be used to display a success message when the player finishes the game. After setting the title and size of the window, we also set resizable to (False, False) to prevent the window from being resized in either direction. This removes any need to re-size the GameScreen images if the player decides to change the window dimensions.

Next we create two PhotoImage objects. These are just tkinter's way of holding an image file in a usable format. These PhotoImages can be placed onto widgets such as Buttons, Labels and Canvases. These two PhotoImages will be going on Buttons which will represent our player's inventory.

We define a Canvas and Frame with fixed widths and heights which allows us to accurately split our screen in two. We use pack_propagate(0) to keep the Frame at its defined size. Frames will shrink to the size necessary to hold their contents by default, but we need this one to stay full-sized irrespective of its children.

We go on to define a StringVar to hold our help text, a Label to display it, another Label to title our inventory, and a second Frame inside the right_frame to hold our inventory items. Our three inventory items are just Buttons which start off showing a question mark image. These are then packed with some padding to space them out a bit. A tuple is used to define (above,below) padding independently (which would be (left,right) inside padx). We stick our inventory items into our inventory_slots list and finish packing before binding a method to left-clicking our canvas.

Handling Game Screens

set_game_screens simply sets a list of GameScreen objects as an attribute of our Game. The reason this isn't in __init__ is because we need a reference to the Game to create the GameScreens.

display_screen takes in an index of our game_screens list and keeps a reference to the GameScreen at that index. It then clears the Canvas and draws our current screen's image onto it. Finally it updates the help Label's StringVar to display its hint to the player.

show_next_screen updates the number which points to our current screen then checks that it is within the bounds of our game_screens. If it is then we display the screen at that index. If it's not then we are out

of screens, indicating that the player has won. In this case we set the `Canvas` to black and show a success message.

Handling Inventory

With this iteration of our inventory system, we're using a `list` to track which slots are available. The length of the `inventory_slots_in_use list` is used to select the next index of our inventory to add a new item to. The same check as `show_next_screen` is used to ensure we are using a valid index of our `inventory_slots` list, and if so the `Button` at that slot is chosen. We `configure` the `Button` with the appropriate `PhotoImage` for the item being added (in this case we just have the key) and append the `item_name` to our `inventory_slots_in_use list` to track that this slot is now in use.

4.1.3 Playing the Game

We begin by making a `Game` object as the main window. We then create three `GameScreens` with their associated image, region-of-interest, item, and hint. The `GameScreen`'s region-of-interest is specified as a 4-tuple with the first two numbers as the top-left x and y, and the second two as the bottom right x and y, forming a rectangle. We merge these together into a `list` and pass it to our `Game` with `set_game_screens`. We finish up by calling `play()` to set the initial screen and `mainloop()` to make the window visible.

4.1.4 Next Iteration

Next up we'll be refining the inventory system logic as well as showing the history of hints in the big space below the current one.

4.2 Our Refined Point-and-Click game

With this iteration our item system is more sophisticated. We can now click and use things from our inventory and specify scenes which require the use of an item to continue. Let's look at how this is done.

```
import tkinter as tk
from tkinter import font
from functools import partial

class GameScreen():
    def __init__(self, master, image, roi, inventory_item=None, help_text=None,
        required_item=None):
        ...
        self.required_item = required_item

    def on_click(self, event, item_in_use):
        if self.master.has_won:
            return

        if item_in_use and not self.required_item:
            self.master.show_cannot_use_message()
        elif (self.roi[0] <= event.x <= self.roi[2]
            and self.roi[1] <= event.y <= self.roi[3]):

            if self.inventory_item:
                self.master.add_inventory_item(self.inventory_item)

            if self.required_item:
                if item_in_use == self.required_item:
                    self.master.show_next_screen()
            else:
                self.master.show_next_screen()
        else:
            if item_in_use:
                self.master.show_cannot_use_message()


class Game(tk.Tk):
    def __init__(self):
        ...
        self.cannot_use_font = font.Font(family="ubuntu", size=28, weight=font.BOLD)
        self.item_in_use = ""
        self.has_won = False

        ...

        self.help_history_var_1 = tk.StringVar(self.right_frame)
        self.help_history_var_2 = tk.StringVar(self.right_frame)
        self.help_history_var_3 = tk.StringVar(self.right_frame)

        help_history_box_1 = tk.Label(self.right_frame, textvar=self.help_history_var_1,
            bg="black", fg="white", padx=10, pady=10)
        help_history_box_2 = tk.Label(self.right_frame, textvar=self.help_history_var_2,
            bg="black", fg="white", padx=10, pady=10)
        help_history_box_3 = tk.Label(self.right_frame, textvar=self.help_history_var_3,
            bg="black", fg="white", padx=10, pady=10)

        help_history_box_1.pack(side=tk.TOP, fill=tk.X, padx=10)
        help_history_box_2.pack(side=tk.TOP, fill=tk.X, padx=10)
        help_history_box_3.pack(side=tk.TOP, fill=tk.X, padx=10)

        ...

        inventory_row_1 = tk.Frame(self.inventory_space, pady=10)
        inventory_row_2 = tk.Frame(self.inventory_space, pady=10)
        inventory_row_3 = tk.Frame(self.inventory_space, pady=10)
```

```
58
59          inventory_slot_1 = tk.Button(self.inventory_row_1,
60                                       image=self.question_mark_image,
61                                       width=50, height=50,
62                                       bg="black",
63                                       command=lambda: self.use_item(0))
64
65          inventory_slot_2 = tk.Button(self.inventory_row_2,
66                                       image=self.question_mark_image,
67                                       width=50, height=50,
68                                       bg="black",
69                                       command=lambda: self.use_item(1))
70
71          inventory_slot_3 = tk.Button(self.inventory_row_3,
72                                            image=self.question_mark_image,
73                                            width=50, height=50,
74                                            bg="black",
75                                            command=lambda: self.use_item(2))
76
77          item_name_1 = tk.StringVar(self.inventory_row_1)
78          item_name_2 = tk.StringVar(self.inventory_row_2)
79          item_name_3 = tk.StringVar(self.inventory_row_3)
80
81          self.item_label_vars = [self.item_name_1, self.item_name_2, self.item_name_3]
82
83          item_label_1 = tk.Label(self.inventory_row_1, textvar=self.item_name_1, anchor="w
                ")
84          item_label_2 = tk.Label(self.inventory_row_2, textvar=self.item_name_2, anchor="w
                ")
85          item_label_3 = tk.Label(self.inventory_row_3, textvar=self.item_name_3, anchor="w
                ")
86
87          inventory_row_1.pack(fill=tk.X, expand=1)
88          inventory_row_2.pack(fill=tk.X, expand=1)
89          inventory_row_3.pack(fill=tk.X, expand=1)
90
91          inventory_slot_1.pack(side=tk.LEFT, padx=(100,20))
92          item_label_1.pack(side=tk.LEFT, fill=tk.X, expand=1)
93          inventory_slot_2.pack(side=tk.LEFT, padx=(100,20))
94          item_label_2.pack(side=tk.LEFT, fill=tk.X, expand=1)
95          inventory_slot_3.pack(side=tk.LEFT, padx=(100,20))
96          item_label_3.pack(side=tk.LEFT, fill=tk.X, expand=1)
97
98          ...
99
100     def handle_click(self, event):
101         ...
102
103     def set_game_screens(self, game_screens):
104         ...
105
106     def display_screen(self, game_screen_number):
107         ...
108         self.show_help_text(self.active_screen.help_text)
109
110     def show_next_screen(self):
111         self.current_screen_number += 1;
112         if self.current_screen_number < len(self.game_screens):
113             self.display_screen(self.current_screen_number)
114             self.clear_used_item()
115         else:
116             self.screen.delete("all")
117             self.screen.configure(bg="black")
118             self.screen.create_text((250,300), text="You Win!", font=self.success_font,
                    fill="white")
119             self.has_won = True
```

```
120
121     def show_help_text(self, text):
122         self.help_history_var_3.set(self.help_history_var_2.get())
123         self.help_history_var_2.set(self.help_history_var_1.get())
124         self.help_history_var_1.set(self.help_var.get())
125         self.help_var.set(text)
126
127     def add_inventory_item(self, item_name):
128         next_available_inventory_slot = len(self.inventory_slots_in_use)
129         if next_available_inventory_slot < len(self.inventory_slots):
130             next_slot = self.inventory_slots[next_available_inventory_slot]
131             next_label_var = self.item_label_vars[next_available_inventory_slot]
132
133             if item_name == "key":
134                 next_slot.configure(image=self.key_image)
135
136             next_label_var.set(item_name.title())
137             self.inventory_slots_in_use.append(item_name)
138
139     def use_item(self, item_number):
140         if item_number < len(self.inventory_slots_in_use):
141             item_name = self.inventory_slots_in_use[item_number]
142             if item_name:
143                 self.item_in_use = item_name
144
145                 for button in self.inventory_slots:
146                     button.configure(bg="black")
147
148                 self.inventory_slots[item_number].configure(bg="white")
149                 self.inventory_slots[item_number].configure(command=self.clear_used_item)
150
151     def clear_used_item(self):
152         self.item_in_use = ""
153         for index, button in enumerate(self.inventory_slots):
154             button.configure(bg="black")
155
156             use_cmd = partial(self.use_item, item_number=index)
157             button.configure(command=use_cmd)
158
159     def show_cannot_use_message(self):
160         text_id = self.screen.create_text((250,25), text="You cannot use that there!",
                font=self.cannot_use_font, fill="white")
161         self.after(2000, lambda: self.screen.delete(text_id))
162
163     def play(self):
164         ...
165
166
167 if __name__ == "__main__":
168     ...
169     scene2 = GameScreen(game, "assets/scene2.png", (117,54,329,412), None, "You Got the
            Key!", "key")
170     ...
```

Listing 4.2: Our Game With Working Inventory

4.2.1 GameScreen

We've now got a new argument for each screen, `required_item`, which establishes whether or not we need to be using an item to advance to the next screen. We've added some new logic to `on_click` to accommodate this.

The method now takes an `item_in_use` argument which represents the active inventory item (if any). First off, we return if the game is won, to prevent clicks on the "You Win" screen. We display a message

to the user if they are trying to use an item on a screen which does not require one, or they are outside of the scene's region-of-interest. When inside the region-of-interest, we check that the `item_in_use` matches the scene's `required_item` and only advance the screen if so. The rest of the logic is the same as before.

4.2.2 Game

`__init__`

We've added a few new attributes to the beginning of our `__init__`. We have a font for the message letting a user know they cannot use their selected item, a string which will hold the item currently in use, and a boolean for whether or not the game has been won.

Afterwards we define three `StringVars` for our help history and 3 `Labels` to hold them. We next need 3 frames to hold our inventory `Buttons` and associated `Labels`. Our `Buttons` have had `commands` added so that they will now use an item when clicked (method will be covered later). We then define three more `StringVars` and `Labels` to display the name of each item next to its button. The `StingVars` are put into a `list` for access later. We finish off by packing everything.

Handling Game Screens

`display_screen` is mostly the same, but now calls a new method `show_help_text` instead of directly manipulating the `help_var`.

`show_next_screen` clears the used item when updating the screen, and sets `has_won` to `True` if the game has displayed all of its screens.

The aforementioned `show_help_text` propagates the values of each of our `help_history StringVars` down to the next one before setting the main `help_var`'s text to that of the current `GameScreen`.

Handling the Inventory

This is where the majority of changes this iteration are. Our `add_inventory_item` method now grabs the `StringVar` in the same index as the next open inventory slot and adds the name of the item to it. The `.title()` here just capitalises the text for aesthetic purposes.

Our new `use_item` method (which is bound to each `Button` in our inventory space) takes in the index of each inventory item as `item_number`, checks it's valid for the size of the `inventory_slots_in_use list` and sets it as our `item_in_use`, which is used by our `GameScreen`'s `on_click`. It then loops through our inventory `Buttons` resetting them to a black background before configuring the clicked `Button` to have a white background, indicating that the item is in use. It also swaps out the `Button`'s `command` to `clear_used_item` so that the user can un-set the item if they want to deactivate it.

Speaking of which, our `clear_used_item` method sets our `item_in_use` to an empty string, resets each `Button`'s background to black and re-binds it's `command` to its previous `use_item`. We need to use a `partial` from the `functools` module to ensure we bind a function with the correct `item_number` argument for each `Button`.

If the player is trying to use an item somewhere in the scene where it is not usable, we need to tell them so. We do this with the `show_cannot_use_message` method. This method creates some text on our `Canvas` with our previously-defined font style. Since the `create_text` method returns a unique ID for the created text, we store that in a variable called `text_id`. We then use tkinter's `after` method to schedule a function to be called after 2 seconds. This function is a `lambda` which deletes the previously-created text by passing its ID to `delete`. This ensures the text does not stay on the player's screen for the rest of that scene.

Playing the Final Game

Just one change here - we pass the `"key"` as the `required_item` to our second scene. This means the player needs to activate the key in their inventory to open the door.

This is where we will leave development of our point-and-click game. The fundamentals of just clicking a region and collecting / using items leads to the potential for a lot of gameplay. A lot of further development would require creating scenes and artwork, which has always been my weakpoint with game development. Despite this I feel like this point-and-click framework has a lot of potential, and is especially interesting given that it is written without an actual game engine.

4.2.3 Further Development

If you'd like to continue work on this project as an exercise, try the following:

- Add a screen which gives the player another item, and a screen which requires this item (how about it's raining outside so the player must pick up an umbrella?).
- Add cutscenes with dialogue boxes which can be advanced by pressing the space bar.
- Add a clues section which has a button for one clue per screen.

Chapter 5

Ini File Editor

In this chapter we'll be creating an app which allows us to edit `.ini` config files. There's a folder in the code repository called `ini_files` with a test file for you to play with while writing out this code. With this project we will learn about the following:

- The `Listbox` widget.
- The `Spinbox` widget.
- Creating a file open and file save dialogue.
- Using keyboard shortcuts with `Menu` items.

5.1 Basic View and Edit Functionality

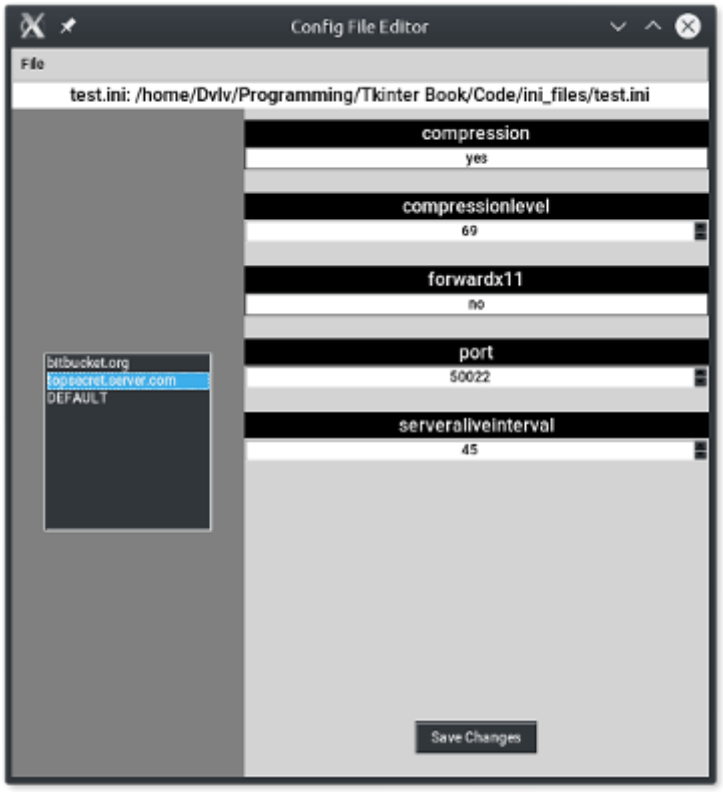

Figure 5.1: Our Ini File Editor

```
import tkinter as tk
from tkinter import filedialog
import tkinter.messagebox as msg
import configparser as cp
import ntpath

class IniEditor(tk.Tk):

    def __init__(self):
        super().__init__()

        self.title("Config File Editor")
        self.geometry("600x600")

        self.active_ini = ""
        self.active_ini_filename = ""
        self.ini_elements = {}

        self.menubar = tk.Menu(self, bg="lightgrey", fg="black")

        self.file_menu = tk.Menu(self.menubar, tearoff=0, bg="lightgrey", fg="black")
        self.file_menu.add_command(label="Open", command=self.file_open, accelerator="
            Ctrl+O")
        self.file_menu.add_command(label="Save", command=self.file_save, accelerator="
            Ctrl+S")

        self.menubar.add_cascade(label="File", menu=self.file_menu)

        self.config(menu=self.menubar)

        self.left_frame = tk.Frame(self, width=200, height=600, bg="grey")
        self.left_frame.pack_propagate(0)

        self.right_frame = tk.Frame(self, width=400, height=600, bg="lightgrey")
        self.right_frame.pack_propagate(0)

        self.file_name_var = tk.StringVar(self)
        self.file_name_label = tk.Label(self, textvar=self.file_name_var, fg="black", bg=
            "white", font=(None, 12))
        self.file_name_label.pack(side=tk.TOP, expand=1, fill=tk.X)

        self.section_select = tk.Listbox(self.left_frame, selectmode=tk.SINGLE)
        self.section_select.configure(exportselection=False)
        self.section_select.pack(expand=1)
        self.section_select.bind("<<ListboxSelect>>", self.display_section_contents)

        self.left_frame.pack(side=tk.LEFT, fill=tk.BOTH)
        self.right_frame.pack(side=tk.LEFT, expand=1, fill=tk.BOTH)

        self.bind("<Control-o>", self.file_open)
        self.bind("<Control-s>", self.file_save)

    def file_open(self, event=None):
        ini_file = filedialog.askopenfilename()

        while ini_file and not ini_file.endswith(".ini"):
            msg.showerror("Wrong Filetype", "Please select an ini file")
            ini_file = filedialog.askopenfilename()

        if ini_file:
            self.parse_ini_file(ini_file)

    def file_save(self, event=None):
        if not self.active_ini:
            msg.showerror("No File Open", "Please open an ini file first")
            return
```

```
64
65          chosen_section = self.section_select.get(self.section_select.curselection())
66
67          for key in self.active_ini[chosen_section]:
68              self.active_ini[chosen_section][key] = self.ini_elements[key].get()
69
70          with open(self.active_ini_filename, "w") as ini_file:
71              self.active_ini.write(ini_file)
72
73          msg.showinfo("Saved", "File Saved Successfully")
74
75      def parse_ini_file(self, ini_file):
76          self.active_ini = cp.ConfigParser()
77          self.active_ini.read(ini_file)
78          self.active_ini_filename = ini_file
79
80          self.section_select.delete(0, tk.END)
81
82          for index, section in enumerate(self.active_ini.sections()):
83              self.section_select.insert(index, section)
84          if "DEFAULT" in self.active_ini:
85              self.section_select.insert(len(self.active_ini.sections()) + 1, "DEFAULT")
86
87          file_name = ": ".join([ntpath.basename(ini_file), ini_file])
88          self.file_name_var.set(file_name)
89
90          self.clear_right_frame()
91
92      def clear_right_frame(self):
93          for child in self.right_frame.winfo_children():
94              child.destroy()
95
96      def display_section_contents(self, event=None):
97          if not self.active_ini:
98              msg.showerror("No File Open", "Please open an ini file first")
99              return
100
101         self.clear_right_frame()
102
103         self.ini_elements = {}
104
105         chosen_section = self.section_select.get(self.section_select.curselection())
106
107         for key in sorted(self.active_ini[chosen_section]):
108             new_label = tk.Label(self.right_frame, text=key, font=(None, 12), bg="black",
                    fg="white")
109             new_label.pack(fill=tk.X, side=tk.TOP, pady=(10,0))
110
111             value = self.active_ini[chosen_section][key]
112
113             if value.isnumeric():
114                 spinbox_default = tk.IntVar(self.right_frame)
115                 spinbox_default.set(int(value))
116                 ini_element = tk.Spinbox(self.right_frame, from_=0, to=99999,
                        textvariable=spinbox_default, bg="white", fg="black", justify="center
                        ")
117             else:
118                 ini_element = tk.Entry(self.right_frame, bg="white", fg="black", justify=
                        "center")
119                 ini_element.insert(0, value)
120
121             ini_element.pack(fill=tk.X, side=tk.TOP, pady=(0,10))
122             self.ini_elements[key] = ini_element
123
124         save_button = tk.Button(self.right_frame, text="Save Changes", command=self.
            file_save)
```

```
125         save_button.pack(side=tk.BOTTOM, pady=(0,20))
126
127
128 if __name__ == "__main__":
129     ini_editor = IniEditor()
130     ini_editor.mainloop()
```

Listing 5.1: Our Ini Editor

5.1.1 __init__

We begin with the hopefully-now-familiar activities such as setting the window title and size, initialising some blank variables, creating our necessary widgets and packing everything. The `active_ini` will hold our parsed `.ini` file, the `active_ini_filename` will hold the name of the given `.ini` file, and `ini_elements` will be used to associate a setting with a `tkinter` widget.

We go on to create a `Menu` containing a "file" option which holds "open" and "save" functionality. The `accelerator` argument passed to the file options is used to display the keyboard shortcut which will activate them.

Our window will be split into two `Frames` with the left being half as big as the right. We again use `pack_propagate(0)` to stop them shrinking. We will also display a `Label` at the top of the window telling the user which file they have open. We specify the `font` argument here to increase the font size. The `font` argument takes a tuple of three: (family, size, style). We can omit the family and style to have the font retain the defaults, and just give the size to make it bigger. This is why we use a tuple of `(None, 12)` to modify only the size to 12.

Now that the layout is sorted, we create the only widget going into our left `Frame` - the `Listbox`. A listbox is somewhat like an expanded dropdown list. It displays multiple elements in a box and allows a user to select them. In this case we only want one selection at a time, so we set the `selectmode` to `tk.SINGLE` to enforce this. We then use `exportselection=False` to prevent the selection from being "lost" when another widget is clicked. We pack it up then bind a method to `<<ListboxSelect>>` so that we can fire off an event when the user selects an option.

After packing our `Frames` we bind the keyboard shortcuts we added to our `Menu` items to the same functions. With that, our `__init__` is complete.

5.1.2 file_open

Our `file_open` method makes use of `tkinter`'s `filedialog` which takes care of opening files for us. The `askopenfilename` method pops up a window with which the user can select a file and returns the path of this file, which we store in `ini_file`. If the given filename does not end with ".ini" we show an error message and bring the open window back up again. We also need to check in this loop condition that `ini_file` is not empty, so that the user can use the "cancel" option to end the interaction. If the filename is valid, we pass it off over to `parse_ini_file`.

5.1.3 parse_ini_file

We begin by creating an instance of a `ConfigParser` which is a library that will handle parsing of `.ini` files into almost-dictionaries. We store this object as `self.active_ini` so that we can refer to it later, then tell it to read and parse the string we got from the file open dialogue. We also store the file path in `self.active_ini_filename` so that we can write to the same file later on.

After opening the file we need to clear any widgets which may still be in our `right_frame`. If we don't do this the user would still see the first file's contents after opening a second, which would be confusing,

and could lead to data loss if they then saved. We achieve this by using `winfo_children` to get all children of the `right_frame` and then calling `destroy` on each to remove it.

Our job now is to get the sections of the file into our `Listbox`. We begin by clearing the `Listbox` in case there are any items left in there from previous file openings. We then `enumerate` over our `.ini` file's sections and `insert` each into our `Listbox`. Since the "DEFAULT" section is not returned by the call to `sections()` we manually account for it afterwards if it exists. We finish up by putting the filename at the top of the window in our `file_name_label`. We use `ntpath` to parse the file name out of the path string, then put a colon, followed by the full path string. Now that our `Listbox` is populated we can display the contents of a section to the user once they have selected one.

5.1.4 display_section_contents

We first need to check we have an `.ini` file to work with, and show an error message if not. We follow on by clearing out the contents of our right `Frame` to ensure it is empty, and then do the same for our dictionary of ini elements. We now need to populate our `Frame` with the elements in the chosen ini section.

The currently selected `Listbox` element is grabbed by passing the id returned by `curselection()` to its `get()` method. Next we iterate over a sorted version of the chosen section in our parsed `.ini` file and create a `Label` with the item's name. The item's value is grabbed using the current `key` and its type is checked. If it's a number, we create a `Spinbox`, otherwise we use a normal `Entry`. The numerical `Spinbox` utilises an `IntVar` (like a `StringVar` for integers) to set its default value to the one read from the ini file. We use the `from_` and `to` arguments to set the minimum and maximum values we can spin to.

We finish off by packing our chosen element and then pairing it with the `key` in our `ini_elements` dictionary. This allows us to keep track of which widget's value should be associated to which config item when saving. Speaking of saving, we also create a `Button` to save without going up into the `Menu`.

5.1.5 file_save

Before we attempt to save we again check to make sure we have a loaded `.ini` file to write to. We then get the chosen section from our `Listbox` and iterate over the section's items. We set each item's value to the value of its associated widget. We finish up by opening the file at the location stored in `active_ini_filename` and telling our `ConfigParser` to write into it. We finally display a message to let the user know that the file has been saved.

5.1.6 Next Iteration

The user currently has to save each section before loading the next one, otherwise any changes will be lost. We'll look at adjusting our `ini_elements` to hold all of the changed values until the program is closed. There's also some graphical tweaks we need to make to better handle the screen resizing.

5.2 Now With Caching and Resizing

With this iteration we hold the updated values in memory even when switching between sections. This means the user can update as many sections as they want and will only need to save once at the end. We've also updated the size of the Frames on re-size. Let's take a look at how this is done:

```
...

class IniEditor(tk.Tk):

    def __init__(self):
        ...
        self.left_frame = tk.Frame(self, width=200, bg="grey")
        self.left_frame.pack_propagate(0)

        self.right_frame = tk.Frame(self, width=400, bg="lightgrey")
        self.right_frame.pack_propagate(0)
        ...
        self.file_name_label.pack(side=tk.TOP, expand=1, fill=tk.X, anchor="n")
        ...
        self.right_frame.bind("<Configure>", self.frame_height)
        ...

    def frame_height(self, event=None):
        new_height = self.winfo_height()
        self.right_frame.configure(height=new_height)

    def file_open(self, event=None):
        ...

    def file_save(self, event=None):
        ...

        for section in self.active_ini:
            for key in self.active_ini[section]:
                try:
                    self.active_ini[section][key] = self.ini_elements[section][key].get()
                except KeyError:
                    # wasn't changed, no need to save it
                    pass

        ...

    def parse_ini_file(self, ini_file):
        ...

        for index, section in enumerate(self.active_ini.sections()):
            self.section_select.insert(index, section)
            self.ini_elements[section] = {}
        if "DEFAULT" in self.active_ini:
            self.section_select.insert(len(self.active_ini.sections()) + 1, "DEFAULT")
            self.ini_elements["DEFAULT"] = {}
        ...

    def clear_right_frame(self):
        ...

    def display_section_contents(self, event):
        if not self.active_ini:
            msg.showerror("No File Open", "Please open an ini file first")
            return

        chosen_section = self.section_select.get(self.section_select.curselection())

        for child in self.right_frame.winfo_children():
            child.pack_forget()
```

```
61
62          for key in sorted(self.active_ini[chosen_section]):
63              new_label = tk.Label(self.right_frame, text=key, font=(None, 12), bg="black",
                    fg="white")
64              new_label.pack(fill=tk.X, side=tk.TOP, pady=(10,0))
65
66              try:
67                  section_elements = self.ini_elements[chosen_section]
68              except KeyError:
69                  section_elements = {}
70
71              try:
72                  ini_element = section_elements[key]
73              except KeyError:
74                  value = self.active_ini[chosen_section][key]
75
76                  if value.isnumeric():
77                      spinbox_default = tk.IntVar(self.right_frame)
78                      spinbox_default.set(int(value))
79                      ini_element = tk.Spinbox(self.right_frame, from_=0, to=99999,
                            textvariable=spinbox_default, bg="white", fg="black", justify="
                            center")
80                  else:
81                      ini_element = tk.Entry(self.right_frame, bg="white", fg="black",
                            justify="center")
82                      ini_element.insert(0, value)
83
84                  self.ini_elements[chosen_section][key] = ini_element
85
86              ini_element.pack(fill=tk.X, side=tk.TOP, pady=(0,10))
87
88          save_button = tk.Button(self.right_frame, text="Save Changes", command=self.
                file_save)
89          save_button.pack(side=tk.BOTTOM, pady=(0,20))
90
91
92  if __name__ == "__main__":
93      ...
```

Listing 5.2: Our Ini Editor

5.2.1 __init__ and frame_height

We've now removed the fixed heights from our `Frames` and bound a method to their `<Configure>` event. This method gets the root window's height and sets the height of the right `Frame` to the same value. The left `Frame` also follows suit. Now when the user re-sizes the window the `Frames` will adjust accordingly. Horizontal adjustment was already handled by the `expand=1` on our right `Frame`'s `pack`.

We have also used the `anchor` argument when packing our `file_name_label` to fix it to the very top of the screen.

5.2.2 parse_ini_file

Since we need to keep track of each individual section's items, we now create an attribute for each section in our `ini_elements` dictionary, which is initialised as another empty dictionary. This will be written to with `display_section_contents`.

5.2.3 display_section_contents

I've left this entire method in for clarity, but some has stayed the same. We now unpack the widgets associated with each section instead of `destroy`ing them so that we can retain a reference to their values.

pack_forget removes widgets from their parent but does not destroy them in memory, meaning we can remove them from the frame without losing their values.

Within our loop we now check to see if we have elements for the chosen section already. If we do we grab them, otherwise we stick an empty dictionary into our section_elements variable to trigger our second except block. If we have the element already, we grab it out of ini_elements and pack it, otherwise we create it, put it into ini_elements, and set the default just as before (except now each element is under the key of its section name). We use try and except to catch KeyErrors here as a way of testing whether or not the elements are already loaded in our cache (ini_elements) rather than as a way of handling something "going wrong". You may know the python idiom "it's easier to ask forgiveness than permission" which is what we have applied here. Instead of trying to check whether or not the ini element has been loaded, we simply assume it has and handle the resulting KeyError if it hasn't.

5.2.4 file_save

Since we now store each element inside the key of its section, we simply iterate over each section and update the active_ini accordingly.

5.2.5 Running

Nothing has changed with regards to running this iteration. You should be able to launch it as before. You can now try changing some of the values under one section, then swapping to a different section and back to the first, and you should see the changes you made have persisted.

5.2.6 Next Iteration

With our current app we can edit existing content but cannot create anything new. We will finish this project off with the ability to create new .ini files, new sections and new items.

5.3 Our finished Ini Editor

Now complete with creating capabilities, let's look at our finalised app:

```
...

class CentralForm(tk.Toplevel):
    def __init__(self, master, my_height=80):
        super().__init__()
        self.master = master

        master_pos_x = self.master.winfo_x()
        master_pos_y = self.master.winfo_y()

        master_width = self.master.winfo_width()
        master_height = self.master.winfo_height()

        my_width = 300

        pos_x = (master_pos_x + (master_width // 2)) - (my_width // 2)
        pos_y = (master_pos_y + (master_height // 2)) - (my_height // 2)

        geometry = "{}x{}+{}+{}".format(my_width, my_height, pos_x, pos_y)
        self.geometry(geometry)


class AddSectionForm(CentralForm):
    def __init__(self, master):
        super().__init__(master)

        self.title("Add New Section")

        self.main_frame = tk.Frame(self, bg="lightgrey")
        self.name_label = tk.Label(self.main_frame, text="Section Name", bg="lightgrey",
            fg="black")
        self.name_entry = tk.Entry(self.main_frame, bg="white", fg="black")
        self.submit_button = tk.Button(self.main_frame, text="Create", command=self.
            create_section)

        self.main_frame.pack(expand=1, fill=tk.BOTH)
        self.name_label.pack(side=tk.TOP, fill=tk.X)
        self.name_entry.pack(side=tk.TOP, fill=tk.X, padx=10)
        self.submit_button.pack(side=tk.TOP, fill=tk.X, pady=(10,0), padx=10)

    def create_section(self):
        section_name = self.name_entry.get()
        if section_name:
            self.master.add_section(section_name)
            self.destroy()
            msg.showinfo("Section Added", "Section " + section_name + " successfully
                added")
        else:
            msg.showerror("No Name", "Please enter a section name", parent=self)


class AddItemForm(CentralForm):
    def __init__(self,  master):

        my_height = 120

        super().__init__(master, my_height)

        self.title("Add New Item")

        self.main_frame = tk.Frame(self, bg="lightgrey")
```

```
59          self.name_label = tk.Label(self.main_frame, text="Item Name", bg="lightgrey", fg=
                "black")
60          self.name_entry = tk.Entry(self.main_frame, bg="white", fg="black")
61          self.value_label = tk.Label(self.main_frame, text="Item Value", bg="lightgrey",
                fg="black")
62          self.value_entry = tk.Entry(self.main_frame, bg="white", fg="black")
63          self.submit_button = tk.Button(self.main_frame, text="Create", command=self.
                create_item)
64
65          self.main_frame.pack(fill=tk.BOTH, expand=1)
66          self.name_label.pack(side=tk.TOP, fill=tk.X)
67          self.name_entry.pack(side=tk.TOP, fill=tk.X, padx=10)
68          self.value_label.pack(side=tk.TOP, fill=tk.X)
69          self.value_entry.pack(side=tk.TOP, fill=tk.X, padx=10)
70          self.submit_button.pack(side=tk.TOP, fill=tk.X, pady=(10,0), padx=10)
71
72      def create_item(self):
73          item_name = self.name_entry.get()
74          item_value = self.value_entry.get()
75          if item_name and item_value:
76              self.master.add_item(item_name, item_value)
77              self.destroy()
78              msg.showinfo("Item Added", item_name + " successfully added")
79          else:
80              msg.showerror("Missing Info", "Please enter a name and value", parent=self)
81
82
83  class IniEditor(tk.Tk):
84
85      def __init__(self):
86          ...
87          self.file_menu = tk.Menu(self.menubar, tearoff=0, bg="lightgrey", fg="black")
88          ...
89          self.bind("<Control-n>", self.file_new)
90          ...
91
92      def add_section_form(self):
93          if not self.active_ini:
94              msg.showerror("No File Open", "Please open an ini file first")
95              return
96
97          AddSectionForm(self)
98
99      def add_section(self, section_name):
100         self.active_ini[section_name] = {}
101         self.populate_section_select_box()
102
103     def frame_height(self, event=None):
104         ...
105
106     def file_new(self, event=None):
107         ini_file = filedialog.asksaveasfilename(filetypes=[("Configuration file", "*.ini"
                )])
108
109         while ini_file and not ini_file.endswith(".ini"):
110             msg.showerror("Wrong Filetype", "Filename must end in .ini")
111             ini_file = filedialog.askopenfilename()
112
113         if ini_file:
114             self.parse_ini_file(ini_file)
115
116     def file_open(self, event=None):
117         ini_file = filedialog.askopenfilename(filetypes=[("Configuration file", "*.ini")
                ])
118         ...
119
```

```
120     def file_save(self, event=None):
121         ...
122 
123     def add_item_form(self):
124         AddItemForm(self)
125 
126     def add_item(self, item_name, item_value):
127         chosen_section = self.section_select.get(self.section_select.curselection())
128         self.active_ini[chosen_section][item_name] = item_value
129         self.display_section_contents()
130 
131     def parse_ini_file(self, ini_file):
132         self.active_ini = cp.ConfigParser()
133         self.active_ini.read(ini_file)
134         self.active_ini_filename = ini_file
135         self.populate_section_select_box()
136 
137         file_name = ": ".join([ntpath.basename(ini_file), ini_file])
138         self.file_name_var.set(file_name)
139 
140         self.clear_right_frame()
141 
142     def clear_right_frame(self):
143         ...
144 
145     def populate_section_select_box(self):
146         self.section_select.delete(0, tk.END)
147 
148         for index, section in enumerate(self.active_ini.sections()):
149             self.section_select.insert(index, section)
150             self.ini_elements[section] = {}
151         if "DEFAULT" in self.active_ini:
152             self.section_select.insert(len(self.active_ini.sections()) + 1, "DEFAULT")
153             self.ini_elements["DEFAULT"] = {}
154 
155     def display_section_contents(self, event=None):
156         ...
157 
158         save_button = tk.Button(self.right_frame, text="Save Changes", command=self.
                file_save)
159         save_button.pack(side=tk.BOTTOM, pady=(0,20))
160 
161         add_button = tk.Button(self.right_frame, text="Add Item", command=self.
                add_item_form)
162         add_button.pack(side=tk.BOTTOM, pady=(0,20))
163 
164 
165 if __name__ == "__main__":
166     ...
```

Listing 5.3: Our Ini Editor

5.3.1 CentralForm

To save a bunch of `__init__` method duplication we've got a base-class for a form which will appear in the center of its parent window. The `__init__` method begins by grabbing the x and y co-ordinates of its master (our IniEditor instance) as well as its width and height. It then has variables representing its own width and height which it uses to calculate where to place itself in order to be in the center of the master and stores these in `pos_x` and `pos_y`. It finally calls the `.geometry()` method on a formatted string of (width x height + x + y) to define its size and position in one go. Now we have this we can create other windows which inherit from this class, and as long as they call `super().__init__(master)` they will be placed in the center of their master. Let's look at our 2 child classes now.

5.3.2 AddSectionForm and AddItemForm

Both of these windows initialise by creating and packing some `Labels` and `Entries` followed by a submit `Button`. Both then have a method attached to their `Button` which grabs the values from the `Entries` and sends them over to the master if they aren't blank. When showing the error `messagebox` we specify the parent as `self` to ensure that it displays on top of our forms. Since the forms and `messageboxes` both display in the center of the master, our error message would appear behind our forms since by default they are parented to the main `Tk` object. By passing in the `parent` argument as `self` we ensure they appear in front of our form. We don't need to do this on success as we destroy the form object beforehand anyway.

5.3.3 IniEditor

`__init__` and file_new

We've now added a "new" option to our file menu and given it a keyboard shortcut. These both call our `file_new` method. This method uses the `asksaveasfilename` method of the `filedialog` to grab a filename from the user, which must end in .ini as before, and then parses it. We've passed in the `filetypes` argument to force the file to end with `.ini` this time (and done the same in `file_open` too). Even thought this new file will be blank, passing it to our `parse_ini_file` method still sets it up in our `active_ini` and `active_ini_filename` variables, as well as putting the filename at the top of our window.

Adding items and sections

Our `add_*_form` methods both just create an instance of the relevant form windows, which then in turn call their `add_*` methods on the master. Our `add_section_form` needs to check there is an .ini file open before running, but our `add_item_form` doesn't need to as the `Button` won't be rendered without an open file.

`add_section` simply adds a new empty dictionary into our `active_ini` with the key matching the text entered into the form. It then calls a new method - `populate_section_select_box` - which clears and re-populates the `Listbox`. It's moved into its own method since we now do this in two places.

`add_item` is similar - but it needs to get the current section from our `Listbox`, and then add a key-value pair to it's dictionary. We then call `display_section_contents` so that the new item appears on the users screen right away and we get its widget into our cache for saving.

display_section_contents

The only change here is to add the "Add Item" `Button` which calls `add_item_form`. Even though we pack this after our `save_button` it will appear above it, due to the use of `tk.BOTTOM`.

That's it for development of our `.ini` file editor. We now have an application which allows us to change specific values without having to wade through the large blocks of comments often written in `.ini` files. Along the way we've learned how to use `Listboxes` to allow the user to make choices which affect the GUI, and `Spinboxes` to allow for precision when adjusting numbers. No more typos when trying to increase a 2 to a 3 and ending up with 23 instead!

5.3.4 Further Development

If you'd like to continue work on this project as an exercise, try the following:

- Make the right `Frame` scrollable using a `Canvas` (remember chapter 2?).
- Alter the running code to allow the user to launch the application with a specific file from the command line, such as "`python inifileeditor.py test.ini`"
- Add deletion functionality to complete all 4 parts of CRUD (Create, Read, Update, Delete).

Chapter 6

A Python Text Editor With Autocomplete and Syntax Highlighting

In this chapter we'll be making a simple Python editor complete with syntax highlighting and some basic auto-completion. Here we'll learn about:

- More advanced features of the `Text` widget.
- More advanced event binding.
- Using `Menus` outside of a top bar.
- Using tags.
- Overriding some of the window manager's event calls.

6.1 Basic Functionality and Autocompletion

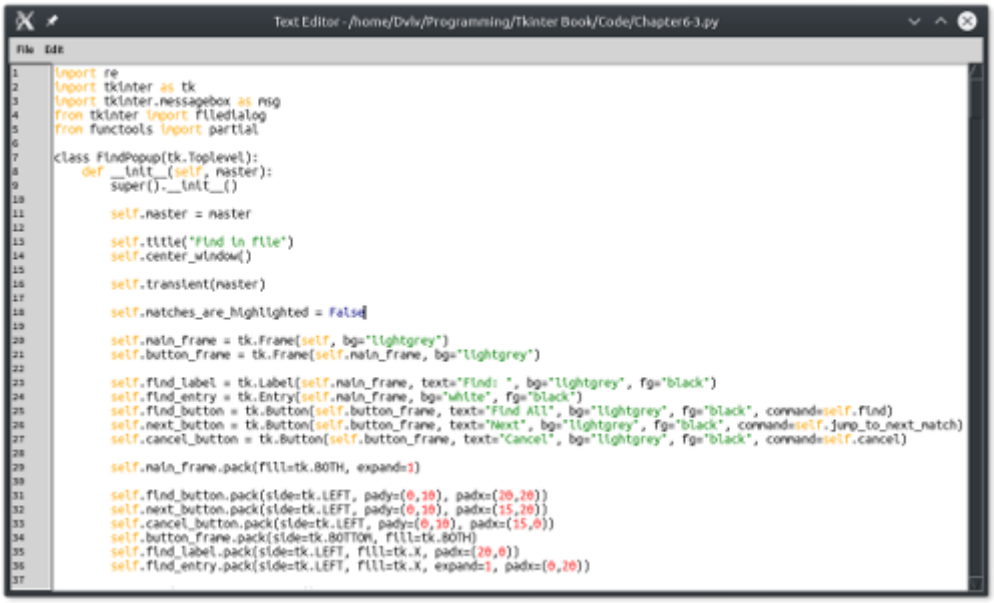

Figure 6.1: Our Text Editor.

```
import tkinter as tk
from tkinter import filedialog
from functools import partial

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

        self.FONT_SIZE = 12
        self.AUTOCOMPLETE_WORDS = ["def", "import", "if", "else", "while", "for","try:",
            "except:", "print(", "True", "False"]
        self.WINDOW_TITLE = "Text Editor"

        self.open_file = ""

        self.title(self.WINDOW_TITLE)
        self.geometry("800x600")

        self.menubar = tk.Menu(self, bg="lightgrey", fg="black")

        self.file_menu = tk.Menu(self.menubar, tearoff=0, bg="lightgrey", fg="black")
        self.file_menu.add_command(label="New", command=self.file_new, accelerator="Ctrl+
            N")
        self.file_menu.add_command(label="Open", command=self.file_open, accelerator="
            Ctrl+O")
        self.file_menu.add_command(label="Save", command=self.file_save, accelerator="
            Ctrl+S")

        self.menubar.add_cascade(label="File", menu=self.file_menu)

        self.configure(menu=self.menubar)

        self.main_text = tk.Text(self, bg="white", fg="black", font=("Ubuntu Mono", self.
            FONT_SIZE))

        self.main_text.pack(expand=1, fill=tk.BOTH)

        self.main_text.bind("<space>", self.destroy_autocomplete_menu)
        self.main_text.bind("<KeyRelease>", self.display_autocomplete_menu)
        self.main_text.bind("<Tab>", self.insert_spaces)

        self.bind("<Control-s>", self.file_save)
        self.bind("<Control-o>", self.file_open)
        self.bind("<Control-n>", self.file_new)

    def file_new(self, event=None):
        file_name = filedialog.asksaveasfilename()
        if file_name:
            self.open_file = file_name
            self.main_text.delete(1.0, tk.END)
            self.title(" - ".join([self.WINDOW_TITLE, self.open_file]))

    def file_open(self, event=None):
        file_to_open = filedialog.askopenfilename()

        if file_to_open:
            self.open_file = file_to_open
            self.main_text.delete(1.0, tk.END)

            with open(file_to_open, "r") as file_contents:
                file_lines = file_contents.readlines()
                if len(file_lines) > 0:
                    for index, line in enumerate(file_lines):
                        index = float(index) + 1.0
                        self.main_text.insert(index, line)

```

```
62          self.title(" - ".join([self.WINDOW_TITLE, self.open_file]))
63
64      def file_save(self, event=None):
65          if not self.open_file:
66              new_file_name = filedialog.asksaveasfilename()
67              if new_file_name:
68                  self.open_file = new_file_name
69
70          if self.open_file:
71              new_contents = self.main_text.get(1.0, tk.END)
72              with open(self.open_file, "w") as open_file:
73                  open_file.write(new_contents)
74
75      def insert_spaces(self, event=None):
76          self.main_text.insert(tk.INSERT, "    ")
77
78          return "break"
79
80      def get_menu_coordinates(self):
81          bbox = self.main_text.bbox(tk.INSERT)
82          menu_x = bbox[0] + self.winfo_x() + self.main_text.winfo_x()
83          menu_y = bbox[1] + self.winfo_y() + self.main_text.winfo_y() + self.FONT_SIZE + 2
84
85          return (menu_x, menu_y)
86
87      def display_autocomplete_menu(self, event=None):
88          current_index = self.main_text.index(tk.INSERT)
89          start = self.adjust_floating_index(current_index)
90
91          try:
92              currently_typed_word = self.main_text.get(start + " wordstart", tk.INSERT)
93          except tk.TclError:
94              currently_typed_word = ""
95
96          currently_typed_word = str(currently_typed_word).strip()
97
98          if currently_typed_word:
99              self.destroy_autocomplete_menu()
100
101             suggestions = []
102             for word in self.AUTOCOMPLETE_WORDS:
103                 if word.startswith(currently_typed_word) and not currently_typed_word ==
                        word:
104                     suggestions.append(word)
105
106             if len(suggestions) > 0:
107                 x, y = self.get_menu_coordinates()
108                 self.complete_menu = tk.Menu(self, tearoff=0, bg="lightgrey", fg="black")
109
110                 for word in suggestions:
111                     insert_word_callback = partial(self.insert_word, word=word, part=
                            currently_typed_word, index=current_index)
112                     self.complete_menu.add_command(label=word, command=
                            insert_word_callback)
113
114                 self.complete_menu.post(x, y)
115                 self.main_text.bind("<Down>", self.focus_menu_item)
116
117     def destroy_autocomplete_menu(self, event=None):
118         try:
119             self.complete_menu.destroy()
120             self.main_text.unbind("<Down>")
121             self.main_text.focus_force()
122         except AttributeError:
123             pass
124
```

```
125     def insert_word(self, word, part, index):
126         amount_typed = len(part)
127         remaining_word = word[amount_typed:]
128         remaining_word_offset = " +" + str(len(remaining_word)) + "c"
129         self.main_text.insert(index, remaining_word)
130         self.main_text.mark_set(tk.INSERT, index + remaining_word_offset)
131         self.destroy_autocomplete_menu()
132         self.main_text.focus_force()
133 
134     def adjust_floating_index(self, number):
135         indices = number.split(".")
136         x_index = indices[0]
137         y_index = indices[1]
138         y_as_number = int(y_index)
139         y_previous = y_as_number - 1
140 
141         return ".".join([x_index, str(y_previous)])
142 
143     def focus_menu_item(self, event=None):
144         try:
145             self.complete_menu.focus_force()
146             self.complete_menu.entryconfig(0, state="active")
147         except tk.TclError:
148             pass
149 
150 if __name__ == "__main__":
151     editor = Editor()
152     editor.mainloop()
```

Listing 6.1: Text Editor

6.1.1 __init__

We begin with some constants. `FONT_SIZE` will be used to adjust the positioning of the autocomplete menu (and also the font size used in our editor, as you probably guessed). Next is the `AUTOCOMPLETE_WORDS` list which holds all of the words which we wish to autocomplete. Finally is the self-explanitory `WINDOW_TITLE`. We also define `open_file` which will be a string representing the path of our currently opened file (much like last chapter) then set the title and geometry.

We then move on to our menu bar, which is much the same as the one from last chapter. We create the new, open, and save buttons which are fairly standard for text editors.

The last thing we need is the main area to enter text, which is achieved using a `Text` widget. We specify the colours and the font (if you don't have ubuntu mono feel free to change this) and pack it to take up as much space as it can with `expand=1` and `fill=tk.BOTH`. We finish up by binding some methods to space, tab, and KeyRelease (each will be covered below) as well as the open, new, and save bindings from our `file_menu`.

6.1.2 Handling Files

`file_new` sets the value of our `open_file` to the one returned by `asksaveasfilename` and empties our `Text` area before changing the window title to display the new file's path.

`file_open` uses `askopenfilename` to grab an existing file name and sets it as our `open_file`. It then clears the contents of our `Text` area to get rid of any existing text in there. Afterwards the file is opened in read mode and we obtain a list of each line with `readlines()`. Each line is `inserted` into our `Text` area at the relevant index. We add 1.0 to the float value of the `list` index because `tkinter`'s indexing starts at 1.0, whereas python's `list` indexing begins at 0. We then finish by displaying the open file in the window's title as before.

`file_save` begins by checking if we have an `open_file`, and if not will try and get one with `asksaveasfilename()`. If that was successful, we grab the text out of our `Text` area and write it into our opened file.

6.1.3 Autocompletion

display_autocomplete_menu

We'll start off with `display_autocomplete_menu` which is bound to `<KeyRelease>`, meaning it's called every time a key is typed into our `Text` area. We begin by grabbing the current index of the cursor with `index(tk.INSERT)`. This is returned in a string of the format "x.y" with x as the line and y as the character offset. For example, the first character of the second line is "2.1" and the 14th character of line 12 is "12.14". The reason we need this is to try and grab the word which is currently being typed by the user. We need to go back one character in order to do this, which is where `adjust_floating_index` comes in. In `adjust_floating_index` we split off the string on the point to get the x and y indices. Then we need to remove 1 from `y_index` and put it back together as a string in the form of "x.y". With this done, we can use `tkinter`'s magic word "wordstart" to get the beginning of the word being typed. This is combined with the `INSERT` position of the cursor to grab the `currently_typed_word`. This may be hard to grasp, so here's a picture which will hopefully clear it up a bit:

Figure 6.2: Finding our current word boundaries. Word is the pink arrow.

Now that we have the currently typed word (or not, if there was a TclError raised along the way due to a bad index) we begin by destroying the autocomplete menu if it is already active, since we will only want one up at a time, and then we build a `list` of suggestions based on the current word. We do this by looping through our `AUTOCOMPLETE_WORDS` and appending ones which start with what the user is currently typing (but not any which are equal to it, since then there's no need to "complete" what they've already typed). If there are any matching suggestions then we need to show the menu. We get the coordinates with `get_menu_coordinates` (covered next) and instantiate a new `Menu` to hold each suggestion.

We loop through each suggestion and create a `partial` of `insert_word` (covered below) passing in the suggested word, the currently-being-typed word and the index of our cursor. We then add a menu item for this word with the `partial` as its command. After all suggestions are added, we use `post(x, y)` to place our menu exactly at the calculated coordinates and bind the down arrow key so that it focuses the first menu item.

get_menu_coordinates

In order to calculate where to put our autocomplete menu we use the `Text` area's `bbox` method to get the bounding box of the cursor position (`tk.INSERT`). We then add on the x and y position of our main window to ensure it displays within the application itself, and add some extra onto the y so that our menu doesn't cover up what the user is currently typing.

insert_word

In order to complete the word being typed, we need to know how much has already been entered. We get this with `len(part)` and use it to get the rest of the word which needs to be inserted. We then need to

build another of tkinter's magic strings to tell it how many characters are being inserted. The format "+nc" implies n characters ahead of the given index, so "+2c" goes 2 characters forward.

With all of that figured out we insert the rest of the word at the current cursor's position and then move the cursor forward the appropriate number of characters with mark_set so that it is at the end of the newly-completed word. We then destroy the autocomplete menu and force the focus back to our Text area so that the user can continue typing.

Focusing and Destroying the Menu

focus_menu_item forces focus onto the autocomplete menu and sets its first item as active so the user can select it with Enter. If we somehow end up here with no menu (or an empty menu) then we will get a TclError, which we can just ignore and do nothing.

destroy_autocomplete_menu calls destroy on our menu and unbinds the down arrow from our Text area. If the menu doesn't exist then the TclError is caught and nothing will happen. We finally force the focus back to our Text area so that the user can continue typing.

6.1.4 Spaces over Tabs!?

There's a method called insert_spaces bound to the Tab key which inserts 4 spaces and uses return "break" to prevent the default behaviour of said key. This is to demonstrate how to make an event binding override the default key behaviour. Using return "break" we end the chain of events caused by pressing the Tab key, meaning no Tab character is inserted. Most editors will offer the option of inserting spaces when pressing Tab, and using 4 spaces conforms to PEP-8.

6.1.5 Next Iteration

Now it's time to utilise some tags to get syntax highlighting working.

6.2 Syntax Highlighting

With this iteration we have some syntax highlighting for strings, numbers, decorators, and various language keywords. A lot of the code has stayed the same, just a small addition to `file_open` to highlight files upon opening them.

```
import re
...

class Editor(tk.Tk):
    def __init__(self):
        ...

        self.AUTOCOMPLETE_WORDS = [
            "def", "import", "as", "if", "elif", "else", "while",
            "for", "try", "except", "print", "True", "False",
            "self", "None", "return", "with"
        ]
        self.KEYWORDS_1 = ["import", "as", "from", "def", "try", "except", "self"]
        self.KEYWORDS_FLOW = ["if", "else", "elif", "try", "except", "for", "in", "while"
            , "return", "with"]

        self.SPACES_REGEX = re.compile("^\s*")
        self.STRING_REGEX_SINGLE = re.compile("'[^'\r\n]*'")
        self.STRING_REGEX_DOUBLE = re.compile('"[^"\r\n]*"')
        self.NUMBER_REGEX = re.compile(r"\b(?=\(*)\d+\.?\d*(?=\)*\,*)\b")
        self.KEYWORDS_REGEX = re.compile("(?=\(*)(?<![a-z])(None|True|False)(?=\)*\,*)")
        self.SELF_REGEX = re.compile("(?=\(*)(?<![a-z])(self)(?=\)*\,*)")
        self.FUNCTIONS_REGEX = re.compile("(?=\(*)(?<![a-z])(print|list|dict|set|int|str)
            (?=\()")

        self.REGEX_TO_TAG = {
            self.STRING_REGEX_SINGLE : "string",
            self.STRING_REGEX_DOUBLE : "string",
            self.NUMBER_REGEX : "digit",
            self.KEYWORDS_REGEX : "keywordcaps",
            self.SELF_REGEX : "keyword1",
            self.FUNCTIONS_REGEX : "keywordfunc",
        }

        ...

        self.main_text.tag_config("keyword1", foreground="orange")
        self.main_text.tag_config("keywordcaps", foreground="navy")
        self.main_text.tag_config("keywordflow", foreground="purple")
        self.main_text.tag_config("keywordfunc", foreground="darkgrey")
        self.main_text.tag_config("decorator", foreground="khaki")
        self.main_text.tag_config("digit", foreground="red")
        self.main_text.tag_config("string", foreground="green")

        ...
        self.main_text.bind("<KeyRelease>", self.on_key_release)
        self.main_text.bind("<Escape>", self.destroy_autocomplete_menu)
        ...

    def file_new(self, event=None):
        ...

    def file_open(self, event=None):
        ...

        final_index = self.main_text.index(tk.END)
        final_line_number = int(final_index.split(".")[0])

        for line_number in range(final_line_number):
            line_to_tag = ".".join([str(line_number), "0"])
```

```
            self.tag_keywords(None, line_to_tag)


    def file_save(self, event=None):
        ...

    def insert_spaces(self, event=None):
        ...

    def get_menu_coordinates(self):
        ...

    def display_autocomplete_menu(self, event=None):
        ...
        self.complete_menu.post(x, y)
        self.complete_menu.bind("<Escape>", self.destroy_autocomplete_menu)
        self.main_text.bind("<Down>", self.focus_menu_item)

    def destroy_autocomplete_menu(self, event=None):
        ...

    def insert_word(self, word, part, index):
        ...

    def adjust_floating_index(self, number):
        ...

    def focus_menu_item(self, event=None):
        ...

    def tag_keywords(self, event=None, current_index=None):
        if not current_index:
            current_index = self.main_text.index(tk.INSERT)
        line_number = current_index.split(".")[0]
        line_beginning = ".".join([line_number, "0"])
        line_text = self.main_text.get(line_beginning, line_beginning + " lineend")
        line_words = line_text.split()
        number_of_spaces = self.number_of_leading_spaces(line_text)
        y_position = number_of_spaces

        for tag in self.main_text.tag_names():
            self.main_text.tag_remove(tag, line_beginning, line_beginning + " lineend")

        self.add_regex_tags(line_number, line_text)

        for word in line_words:
            stripped_word = word.strip("():,")
            word_start = str(y_position)
            word_end = str(y_position + len(stripped_word))
            start_index = ".".join([line_number, word_start])
            end_index = ".".join([line_number, word_end])

            if stripped_word in self.KEYWORDS_1:
                self.main_text.tag_add("keyword1", start_index, end_index)
            elif stripped_word in self.KEYWORDS_FLOW:
                self.main_text.tag_add("keywordflow", start_index, end_index)
            elif stripped_word.startswith("@"):
                self.main_text.tag_add("decorator", start_index, end_index)

            y_position += len(word) + 1

    def number_of_leading_spaces(self, line):
        spaces = re.search(self.SPACES_REGEX, line)
        if spaces.group(0) is not None:
            number_of_spaces = len(spaces.group(0))
        else:
```

```
125             number_of_spaces = 0
126
127         return number_of_spaces
128
129     def add_regex_tags(self, line_number, line_text):
130         for regex, tag in self.REGEX_TO_TAG.items():
131             for match in regex.finditer(line_text):
132                 start, end = match.span()
133                 start_index = ".".join([line_number, str(start)])
134                 end_index = ".".join([line_number, str(end)])
135                 self.main_text.tag_add(tag, start_index, end_index)
136
137     def on_key_release(self, event=None):
138         if not event.keysym in ("Up", "Down", "Left", "Right", "BackSpace", "Delete", "
                Escape"):
139             self.display_autocomplete_menu()
140         self.tag_keywords()
141
142 if __name__ == "__main__":
143     ...
```

Listing 6.2: Text Editor

6.2.1 __init__

We've got some more autocomplete words now as well as two more lists which separate them out a bit. This is to avoid colouring all keywords with the same colour, which looks horrible in my opinion. We then have a big pile of regexes which will match spaces, strings, numbers and keywords. I will try to explain each below. After that we've got a dictionary mapping the regexes to strings, which are some of the tag names defined below. We use `tag_config` to define a tag represented by a string (the first argument) and add some styling associated with it (the proceeding keyword arguments). Anything which is given the tag "keyword1" will be orange, for example.

A tag is essentially just a group of properties which can be assigned to certain characters within the `Text` area. In this instance we are changing the colour of certain words to achieve syntax highlighting.

We've adjusted the method bound to `<KeyRelease>` to a new one, since we now want to call 2 methods each time. This will be covered later.

6.2.2 Regexes Explained

```
1  self.STRING_REGEX_SINGLE = "'[^'\r\n]*'"
2  # a literal '
3  # anything which isn't ' or a newline 0 or more times
4  # a literal '
5
6  self.STRING_REGEX_DOUBLE = re.compile('"[^"\r\n]*"')
7  # a literal ",
8  # anything which isn't " or a newline 0 or more times
9  # a literal "
10
11 self.NUMBER_REGEX = re.compile(
12     \b          # begin with a word boundry (punctuation or space)
13     (?=\(*)     # match but don't highlight 0 or more opening brackets
14     \d+\.?\d*   # match 1 or more numbers, 0 or 1 decimal points, 0 or more numbers
15     (?=\)*\,*)  # match but don't highlight 0 or more closing brackets or commas
16     \b          # end with a word boundry (punctuation or space)
17 )
18
19
20 self.KEYWORDS_REGEX = re.compile(
```

```
21      (?=\(*)           # match but don't highlight 0 or more opening brackets
22      (?<![a-z])        # don't match if it begins with an alphabet character
23      (None|True|False) # match None or True or False
24      (?=\)*\,*)        # match but don't highlight 0 or more closing brackets or commas
25  )
26
27  self.SELF_REGEX = re.compile(
28      (?=\(*)    # same as above
29      (?<![a-z]) # same as above
30      (self)     # match self
31      (?=\)*\,*) # same as above
32  )
33
34
35  self.FUNCTIONS_REGEX = re.compile(
36      (?=\(*)                       # same as above
37      (?<![a-z])                    # same as above
38      (print|list|dict|set|int|str) # literal match print, list, dict, etc.
39      (?=\()                        # match but dont capture 1 opening bracket
40  )
```

Listing 6.3: Regex Explanations

6.2.3 file_open

After all of the previous code for opening files, we need to run them through our `tag_keywords` method to apply the syntax highlighting. Since this function works line-by-line, we get the index of the end of our file and split the x off of `tkinter`'s "x.y" indexing format. This gives us the number of the last line, which is also the number of lines in the document. We can then iterate over the range of that number, build a `tkinter` index of "line_number.0" and pass it into our `tag_keywords` method. Speaking of which:

6.2.4 tag_keywords

The main bulk of this iteration is right here. As mentioned, this method works on a line-by-line basis, so we need to check whether we have a line number passed in. If not, we use the line with the cursor on it. We again split off the x and join it with a 0 to get the `tkinter` index of the line's beginning. We combine that with the magic word "lineend" within `get` to get the contents of the whole line. We can then use `split()` to get each individual "word" on the line. We grab the number of leading spaces on the line so that we can adjust our y position to the start of the actual text.

With all of that set up, we remove all tags on the current line so that we can overwrite them with new ones. We do this by looping through all of our `tag_names()` and calling `tag_remove` on the entire line. Without this, when the user types "as" it will become highlighted because it is a keyword. If they then continue to write the full word "assumption" the first "as" will remain highlighted, which will look wrong and be offputting.

The first thing to do is to add the regex-specified tags. Let's jump to that method now:

add_regex_tags

We iterate over our `dictionary` of regex-to-tag mappings and use `find_iter` over the current line to see if we have any matches. If we do, the `span()` function handily gives us the start and end indexes of the entire string at which this match occcurs. We `join` these to the line number with a dot to match `tkinter`'s indexing and add the associated tag in that range.

back to tag_keywords

Now that we've covered the more complex cases we can do a slightly more manual approach to finish off the remaining keyword types. We strip off brackets, colons, and commas because they are part of some keywords (if:, else:) but we don't want them to be coloured. We then use the current y position as the

word's start and add the length to it to get the word's end. We join it with the line number to get an index as usual so that we can begin comparison.

All we have to do is check whether the word is in one of our keywords `lists`, and if it is, assign the relevant tag to its range. We just use `startswith("@")` to find a decorator for simplicity. We then update the current y position with the length of the word plus one (for the space character).

That's all there is to applying the syntax highlighting to our `Text` area. The majority of the work is figuring out how to correctly keep track of the relevant `tkinter` index of the word you wish to colour.

Why Two Methods of Tagging?

Certain keywords should not be observed as part of a bigger "word". Take "if" for example. It should generally appear by itself (aside from the colon, which we can easily strip off). Now consider "None". "None" will often get merged into a bigger "word". For example: `self.add_task(None, task_text, True)`. Here there is no spacing around "None", which is the correct python styling, but when `splitting` this line we get one big chunk of `self.add_task(None,` which is not equal to "None". We can't pick out the "None" easily here, which is why we need to use regex.

Strings and numbers are also different beasts entirely. You can't really build a list of all possible strings or numbers, so regex is a must in order to match them.

6.2.5 display_autocomplete_menu, number_of_leading_spaces, and on_key_release

`display_autocomplete_menu` now has `destroy_autocomplete_menu` bound to Escape so that the user can close it and continue typing. The same binding was added to our `main_text` in `__init__`.

`number_of_leading_spaces` is a method taken from an older project of mine. It uses a regex matching 0 or more space characters at the start of a string. If it finds a match, we return the length of the match, otherwise 0.

`on_key_release` is just created to call two methods on the `<KeyRelease>` event. It displays the autocomplete menu as before as well as updating our syntax highlighting tags with `tag_keywords`. We do not want to display the autocomplete menu on a few specific key presses, including the arrow keys, backspace, and escape, so we will check the `event.keysym` before calling `display_autocomplete_menu`. `event.keysym` returns a human-readable representation of the key which triggered the event.

6.2.6 Next Iteration

We'll finish off our text editor by adding some standard features to bring it in line with other text editors, including a scroll bar, line numbers, select-all, find, and an Edit menu.

6.3 Our Finished Editor

```
...
import tkinter.messagebox as msg

class FindPopup(tk.Toplevel):
    def __init__(self, master):
        super().__init__()

        self.master = master

        self.title("Find in file")
        self.center_window()

        self.transient(master)

        self.matches_are_highlighted = True

        self.main_frame = tk.Frame(self, bg="lightgrey")
        self.button_frame = tk.Frame(self.main_frame, bg="lightgrey")

        self.find_label = tk.Label(self.main_frame, text="Find: ", bg="lightgrey", fg="
            black")
        self.find_entry = tk.Entry(self.main_frame, bg="white", fg="black")
        self.find_button = tk.Button(self.button_frame, text="Find All", bg="lightgrey",
            fg="black", command=self.find)
        self.next_button = tk.Button(self.button_frame, text="Next", bg="lightgrey", fg="
            black", command=self.jump_to_next_match)
        self.cancel_button = tk.Button(self.button_frame, text="Cancel", bg="lightgrey",
            fg="black", command=self.cancel)

        self.main_frame.pack(fill=tk.BOTH, expand=1)

        self.find_button.pack(side=tk.LEFT, pady=(0,10), padx=(20,20))
        self.next_button.pack(side=tk.LEFT, pady=(0,10), padx=(15,20))
        self.cancel_button.pack(side=tk.LEFT, pady=(0,10), padx=(15,0))
        self.button_frame.pack(side=tk.BOTTOM, fill=tk.BOTH)
        self.find_label.pack(side=tk.LEFT, fill=tk.X, padx=(20,0))
        self.find_entry.pack(side=tk.LEFT, fill=tk.X, expand=1, padx=(0,20))

        self.find_entry.focus_force()
        self.find_entry.bind("<Return>", self.jump_to_next_match)
        self.find_entry.bind("<KeyRelease>", self.matches_are_not_highlighted)
        self.bind("<Escape>", self.cancel)

        self.protocol("WM_DELETE_WINDOW", self.cancel)

    def find(self, event=None):
        text_to_find = self.find_entry.get()
        if text_to_find and not self.matches_are_highlighted:
            self.master.remove_all_find_tags()
            self.master.highlight_matches(text_to_find)
            self.matches_are_highlighted = True

    def jump_to_next_match(self, event=None):
        text_to_find = self.find_entry.get()
        if text_to_find:
            if not self.matches_are_highlighted:
                self.find()
            self.master.next_match()

    def cancel(self, event=None):
        self.master.remove_all_find_tags()
        self.destroy()

    def matches_are_not_highlighted(self, event):
```

```
61          key_pressed = event.keysym
62          if not key_pressed == "Return":
63              self.matches_are_highlighted = False
64
65      def center_window(self):
66          master_pos_x = self.master.winfo_x()
67          master_pos_y = self.master.winfo_y()
68
69          master_width = self.master.winfo_width()
70          master_height = self.master.winfo_height()
71
72          my_width = 300
73          my_height = 100
74
75          pos_x = (master_pos_x + (master_width // 2)) - (my_width // 2)
76          pos_y = (master_pos_y + (master_height // 2)) - (my_height // 2)
77
78          geometry = "{}x{}+{}+{}".format(my_width, my_height, pos_x, pos_y)
79          self.geometry(geometry)
80
81
82
83  class Editor(tk.Tk):
84      def __init__(self):
85          ...
86          self.edit_menu = tk.Menu(self.menubar, tearoff=0, bg="lightgrey", fg="black")
87          self.edit_menu.add_command(label="Cut", command=self.edit_cut, accelerator="Ctrl+
                X")
88          self.edit_menu.add_command(label="Paste", command=self.edit_paste, accelerator="
                Ctrl+V")
89          self.edit_menu.add_command(label="Undo", command=self.edit_undo, accelerator="
                Ctrl+Z")
90          self.edit_menu.add_command(label="Redo", command=self.edit_redo, accelerator="
                Ctrl+Y")
91
92          self.menubar.add_cascade(label="File", menu=self.file_menu)
93          self.menubar.add_cascade(label="Edit", menu=self.edit_menu)
94
95          ...
96
97          self.line_numbers = tk.Text(self, bg="lightgrey", fg="black", width=6)
98          self.line_numbers.insert(1.0, "1 \n")
99          self.line_numbers.configure(state="disabled")
100         self.line_numbers.pack(side=tk.LEFT, fill=tk.Y)
101
102         ...
103
104         self.scrollbar = tk.Scrollbar(self, orient="vertical", command=self.
                scroll_text_and_line_numbers)
105         self.main_text.configure(yscrollcommand=self.scrollbar.set)
106
107         self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y)
108         self.main_text.pack(expand=1, fill=tk.BOTH)
109
110         ...
111         self.main_text.tag_config("findmatch", background="yellow")
112
113         ...
114
115         self.main_text.bind("<Control-y>", self.edit_redo)
116
117         ...
118
119         self.bind("<Control-a>", self.select_all)
120         self.bind("<Control-f>", self.show_find_window)
121
```

```
122         self.main_text.bind("<MouseWheel>", self.scroll_text_and_line_numbers)
123         self.main_text.bind("<Button-4>", self.scroll_text_and_line_numbers)
124         self.main_text.bind("<Button-5>", self.scroll_text_and_line_numbers)
125 
126         self.line_numbers.bind("<MouseWheel>", self.skip_event)
127         self.line_numbers.bind("<Button-4>", self.skip_event)
128         self.line_numbers.bind("<Button-5>", self.skip_event)
129 
130     def skip_event(self, event=None):
131         return "break"
132 
133     def scroll_text_and_line_numbers(self, *args):
134         try:
135             # from scrollbar
136             self.main_text.yview_moveto(args[1])
137             self.line_numbers.yview_moveto(args[1])
138         except IndexError:
139             #from MouseWheel
140             event = args[0]
141             if event.delta:
142                 move = -1*(event.delta/120)
143             else:
144                 if event.num == 5:
145                     move = 1
146                 else:
147                     move = -1
148 
149             self.main_text.yview_scroll(int(move), "units")
150             self.line_numbers.yview_scroll(int(move), "units")
151 
152         return "break"
153 
154     def file_new(self, event=None):
155         ...
156 
157     def file_open(self, event=None):
158         file_to_open = filedialog.askopenfilename()
159 
160         if file_to_open:
161             self.open_file = file_to_open
162             self.main_text.delete(1.0, tk.END)
163 
164             with open(file_to_open, "r") as file_contents:
165                 file_lines = file_contents.readlines()
166                 if len(file_lines) > 0:
167                     for index, line in enumerate(file_lines):
168                         index = float(index) + 1.0
169                         self.main_text.insert(index, line)
170 
171         self.title(" - ".join([self.WINDOW_TITLE, self.open_file]))
172 
173         self.tag_all_lines()
174 
175 
176     def file_save(self, event=None):
177         ...
178 
179     def select_all(self, event=None):
180         self.main_text.tag_add("sel", 1.0, tk.END)
181 
182         return "break"
183 
184     def edit_cut(self, event=None):
185         self.main_text.event_generate("<<Cut>>")
186 
187         return "break"
```

```
188
189     def edit_paste(self, event=None):
190         self.main_text.event_generate("<<Paste>>")
191         self.on_key_release()
192         self.tag_all_lines()
193
194         return "break"
195
196     def edit_undo(self, event=None):
197         self.main_text.event_generate("<<Undo>>")
198
199         return "break"
200
201     def edit_redo(self, event=None):
202         self.main_text.event_generate("<<Redo>>")
203
204         return "break"
205
206     def insert_spaces(self, event=None):
207         ...
208
209     def get_menu_coordinates(self):
210         ...
211
212     def display_autocomplete_menu(self, event=None):
213         ...
214
215     def destroy_autocomplete_menu(self, event=None):
216         ...
217
218     def insert_word(self, word, part, index):
219         ...
220
221     def adjust_floating_index(self, number):
222         ...
223
224     def focus_menu_item(self, event=None):
225         ...
226
227     def tag_keywords(self, event=None, current_index=None):
228         ...
229
230     def number_of_leading_spaces(self, line):
231         ...
232
233     def add_regex_tags(self, line_number, line_text):
234         ...
235
236     def on_key_release(self, event=None):
237         ...
238         self.update_line_numbers()
239
240     def tag_all_lines(self):
241         final_index = self.main_text.index(tk.END)
242         final_line_number = int(final_index.split(".")[0])
243
244         for line_number in range(final_line_number):
245             line_to_tag = ".".join([str(line_number), "0"])
246             self.tag_keywords(None, line_to_tag)
247
248         self.update_line_numbers()
249
250     def update_line_numbers(self):
251         self.line_numbers.configure(state="normal")
252         self.line_numbers.delete(1.0, tk.END)
253         number_of_lines = self.main_text.index(tk.END).split(".")[0]
```

```
254         line_number_string = "\n".join(str(no+1) for no in range(int(number_of_lines)))
255         self.line_numbers.insert(1.0, line_number_string)
256         self.line_numbers.configure(state="disabled")
257 
258     def show_find_window(self, event=None):
259         FindPopup(self)
260 
261     def highlight_matches(self, text_to_find):
262         self.main_text.tag_remove("findmatch", 1.0, tk.END)
263         self.match_coordinates = []
264         self.current_match = -1
265 
266         find_regex = re.compile(text_to_find)
267         search_text_lines = self.main_text.get(1.0, tk.END).split("\n")
268 
269         for line_number, line in enumerate(search_text_lines):
270             line_number += 1
271             for match in find_regex.finditer(line):
272                 start, end = match.span()
273                 start_index = ".".join([str(line_number), str(start)])
274                 end_index = ".".join([str(line_number), str(end)])
275                 self.main_text.tag_add("findmatch", start_index, end_index)
276                 self.match_coordinates.append((start_index, end_index))
277 
278     def next_match(self, event=None):
279         try:
280             current_target, current_target_end = self.match_coordinates[self.
                current_match]
281             self.main_text.tag_remove("sel", current_target, current_target_end)
282             self.main_text.tag_add("findmatch", current_target, current_target_end)
283         except IndexError:
284             pass
285 
286         try:
287             self.current_match = self.current_match + 1
288             next_target, target_end = self.match_coordinates[self.current_match]
289         except IndexError:
290             if len(self.match_coordinates) == 0:
291                 msg.showinfo("No Matches", "No Matches Found")
292             else:
293                 if msg.askyesno("Wrap Search?", "Reached end of file. Continue from the
                    top?"):
294                     self.current_match = -1
295                     self.next_match()
296         else:
297             self.main_text.mark_set(tk.INSERT, next_target)
298             self.main_text.tag_remove("findmatch", next_target, target_end)
299             self.main_text.tag_add("sel", next_target, target_end)
300             self.main_text.see(next_target)
301 
302     def remove_all_find_tags(self):
303         self.main_text.tag_remove("findmatch", 1.0, tk.END)
304         self.main_text.tag_remove("sel", 1.0, tk.END)
305 
306 
307 if __name__ == "__main__":
308     editor = Editor()
309     editor.mainloop()
```

Listing 6.4: Our Finished Editor

6.3.1 FindPopup

__init__

After setting the title and borrowing code from our ini editor to center this window with the `center_window` method, we specify that this window should be a `transient`, which means it will remain over the top of

our main window until closed. Next is a boolean which we use to indicate if the matches are highlighted in the main window or not. We then define two frames: a main one for the whole window and a button frame to hold our Buttons. We pack our Label and Entry in the main_frame and our three Buttons - Find All, Next, and Cancel - into the button_frame, which is packed to the bottom of the main_frame. We force focus to the Entry so that the user doesn't have to click in it to begin typing, bind Enter to our jump_to_next_match method, bind Escape to our cancel method, and override the window manager using self.protocol("WM_DELETE_WINDOW", <callback>) so that our cancel method will be called when the user closes the window.

The rest

Our find method sets the matches_are_highlighted flag to True to avoid repeatedly calling the highlight_matches method of the master window, and calls highlight_matches with the text from our Entry, providing there is something written in there and the matches are not already highlighted.

jump_to_next_match will call find() if the matches for the Entry's text are not currently highlighted, then pass off to the next_match method of our master window.

cancel will tell the master window to remove the tags added by the find methods and then destroy our FindPopup instance.

matches_are_not_highlighted will set matches_are_highlighted to False if any key except Enter is pressed within our Entry, as this indicates the word to search for has now changed and needs to be re-found.

center_window came from our Ini Editor, so see the previous chapter for an explanation.

6.3.2 Editor

__init__

With this iteration, we have an edit menu to accompany our file menu. It's created in the same way with cut, paste, undo, and redo buttons.

Our line numbers are handled by a disabled Text widget. It's six characters wide, meaning it can keep track of up to one million lines of code (I hope nobody ever encounters a million-line file however!) We start it off at line 1 and pack it over to the left.

We create a Scrollbar and bind it to a command - scroll_text_and_line_numbers - as it will need to scroll both of our Text widgets simultaneously. We also pair the main_text's yscrollcommand to the bar to ensure the bar moves when we scroll with the mouse. We pack this to the right before finally packing our main_text so that everything is in the right place.

We finish up by adding a new tag - findmatch - to indicate matches made from our FindPopup, and finally binding some key events.

Scrolling

scroll_text_and_line_numbers will receive different arguments depending on if it is triggered by the Scrollbar or mouse wheel. The Scrollbar will pass a tuple of ("moveto", <fraction>) over here, so we can directly call yview_moveto and pass over the fraction argument. Our mouse wheel will only pass the usual event object which will raise an IndexError if we try and grab element [1] from it. Therefore we catch this exception and use the code we saw in Chapter 2 to scroll both areas.

Our skip_event method is bound to the mouse wheel on the line_numbers. This is to stop the user from scrolling the line numbers. The method just uses return "break" in order to do nothing but end the

chain of events triggered by scrolling.

select_all, file_open, and on_key_release

Simple changes here. We want to update the line numbers after opening a file for obvious reasons, so we call `update_line_numbers` (covered later). Same deal for `on_key_release`. `select_all` adds the `"sel"` tag to all of the text in our `main_text` area, thereby selecting it all.

The Edit Menu

As well as binding callbacks to events in `tkinter`, we can generate the events ourselves using `event_generate`. Here we generate the `<<Cut>>`, `<<Paste>>`, `<<Undo>>`, and `<<Redo>>` events.

After pasting we want to make sure the new text is syntax-highlighted. To do this we have abstracted some code from `file_open` into a new function - `tag_all_lines` - which we call after pasting. We also call `on_key_release` directly, since we are returning `"break"`, which will both update the line numbers and trigger auto-completion if we paste part of a keyword. We have bound `<Control-v>` to this paste method in `__init__` to ensure this happens when the user pastes from the keyboard shortcut too.

update_line_numbers

In order to update the line numbers as the opened file grows, we enable our `line_numbers` widget, remove all of its contents, grab the number of lines off of the end-of-file index, join each number in the range up to our final line with a newline character, place this long string into the widget, and finally disable it again. Note that we add 1 to each line number in our loop. This is because we *don't* want our first line to be line 0 and we *do* want the last value included.

highlight_matches

We begin this method by removing all "findmatch" tags from our `main_text` widget and initialising a couple of variables which we will use to keep track of our matches. We then compile `text_to_match`, which came from the `Entry` in our FindPopup, as regex. This allows the user to put an actual regular expression in this box as well as the literal text. We then split the `main_text`'s contents on newline characters to get a `list` of every line. We `enumerate` over this `list` and use code very similar to that in our `add_regex_tags` to add a "findmatch" tag to the relevant `tkinter` index range containing our matches. We need to add 1 to the `line_number` when enumerating because a `list` index begins at 0 but a `tkinter` line number index begins at 1.

next_match

This method makes use of `current_match` and `match_coordinates` which were both initialised and built in our `highlight_matches` method. We begin by trying to remove the currently selected match's "sel" tag so that we only have one match selected at a time. If there isn't one we will get an `IndexError` which we will just catch and `pass`.

We then increment our `current_match` by 1 and try to grab the next set of match coordinates. If this also throws an `IndexError` then we either have no matches or we are at the final match of the file. If the `len` of our `match_coordinates` list is 0 then we have no matches, so we will show a `messagebox` letting the user know. Otherwise we are at the final match in the file, so we use an `askyesno` to ask the user if they want to wrap the search back to the top. If they choose "yes" we put `current_match` back to -1 and re-run this `next_match` method.

If no error is caught we put the cursor at the start of the matched word, swap its "findmatch" tag for the "sel" tag to select it, then use `see` to scroll the `main_text` widget enough so that the match comes into view.

6.3.3 The Finished Product

We've now got a nice little text editor with some syntax highlighting, autocomplete, and a find menu, along with a few standard features you would expect to be in a text editor. I'm going to leave this chapter here, even though there are so many more things I think can be added to this project, and it's really tempting to just carry on forever. Feel free to play with this project to really customise it to your own preferences, everything from colour schemes to keyboard shortcuts. Hopefully from writing this code you will have learned how powerful a tool the tags are within `tkinter`, and gained an understanding of how `tkinter` keeps track of indexing.

The `Text` widget provides a `search` method of its own which can be used to obtain indexes of any matches, and supports regexes. I decided to stick with manual regex searching and processing using `find_iter` and constructing the `tkinter` indexes to better show how they work. If you wish to re-write some of the code to practise using the `search` method, please do.

6.3.4 Further Development

If you'd like to continue work on this project as an exercise, try the following:

- Use a checkbox or radio buttons to give the user the option of using either regex or plain-text search with the FindPopup.
- Utilise the `colorchooser` widget to give the user the ability to change some of the colour scheme.
- Add Replace functionality to the FindPopup.
- Use regex to pull all of the function names from the opened file and provide a popup window to list them all.
- Pick any feature you like from a text editor and try to implement it.

Chapter 7

A Pomodoro Timer

In this chapter we will be creating an app which will help people to follow the pomodoro technique. The pomodoro technique involves concentrating on a task for 25 minute bursts, so we will be building a timer which will count down for 25 minutes then alert the user when the time is up. It will also contain a log of completed tasks. In this chapter we will learn about the following:

- Using threads with tkinter
- the ttk Treeview widget
- Using ttk widgets for a more native look

7.1 A Basic Timer

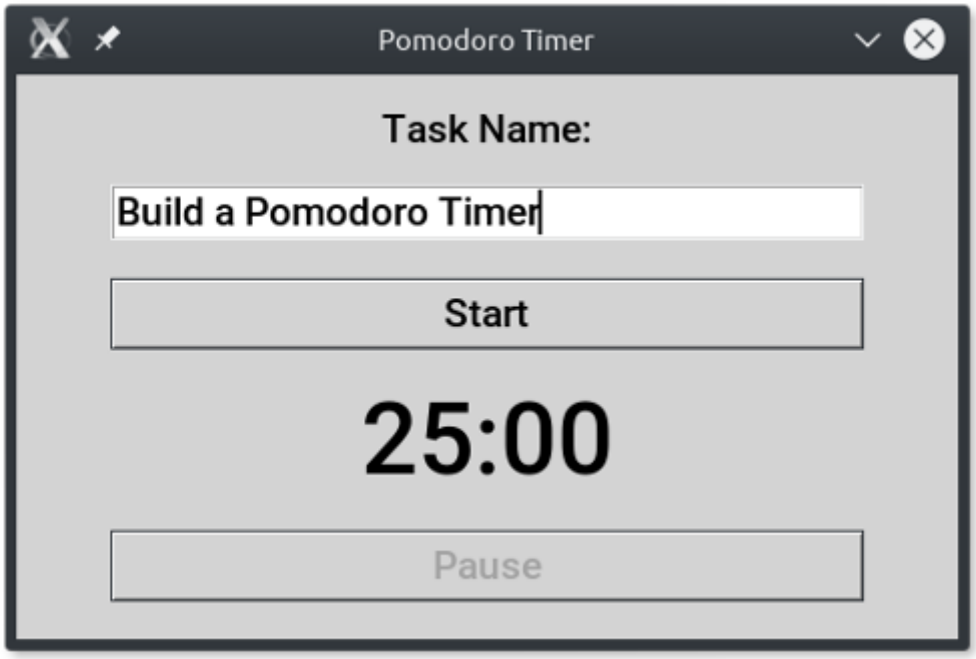

Figure 7.1: A Pomodoro Timer

```
1 import threading
2 import time
3 import datetime
4 import tkinter as tk
5 from tkinter import messagebox as msg
6
```

```
7  class CountingThread(threading.Thread):
8      def __init__(self, master, start_time, end_time):
9          super().__init__()
10         self.master = master
11         self.start_time = start_time
12         self.end_time = end_time
13
14         self.end_now = False
15         self.paused = False
16         self.force_quit = False
17
18     def run(self):
19         while True:
20             if not self.paused and not self.end_now and not self.force_quit:
21                 self.main_loop()
22                 if datetime.datetime.now() >= self.end_time:
23                     if not self.force_quit:
24                         self.master.finish()
25                         break
26             elif self.end_now:
27                 self.master.finish()
28                 break
29             elif self.force_quit:
30                 del self.master.worker
31                 return
32             else:
33                 continue
34         return
35
36     def main_loop(self):
37         now = datetime.datetime.now()
38         if now < self.end_time:
39             time_difference = self.end_time - now
40             mins, secs = divmod(time_difference.seconds, 60)
41             time_string = "{:02d}:{:02d}".format(mins, secs)
42             if not self.force_quit:
43                 self.master.update_time_remaining(time_string)
44
45
46 class Timer(tk.Tk):
47     def __init__(self):
48         super().__init__()
49
50         self.title("Pomodoro Timer")
51         self.geometry("500x300")
52         self.resizable(False, False)
53
54         self.standard_font = (None, 16)
55
56         self.main_frame = tk.Frame(self, width=500, height=300, bg="lightgrey")
57
58         self.task_name_label = tk.Label(self.main_frame, text="Task Name:", bg="lightgrey
               ", fg="black", font=self.standard_font)
59         self.task_name_entry = tk.Entry(self.main_frame, bg="white", fg="black", font=
               self.standard_font)
60         self.start_button = tk.Button(self.main_frame, text="Start", bg="lightgrey", fg="
               black", command=self.start, font=self.standard_font)
61         self.time_remaining_var = tk.StringVar(self.main_frame)
62         self.time_remaining_var.set("25:00")
63         self.time_remaining_label = tk.Label(self.main_frame, textvar=self.
               time_remaining_var, bg="lightgrey", fg="black", font=(None, 40))
64         self.pause_button = tk.Button(self.main_frame, text="Pause", bg="lightgrey", fg="
               black", command=self.pause, font=self.standard_font, state="disabled")
65
66         self.main_frame.pack(fill=tk.BOTH, expand=1)
67
```

```
68          self.task_name_label.pack(fill=tk.X, pady=15)
69          self.task_name_entry.pack(fill=tk.X, padx=50, pady=(0,20))
70          self.start_button.pack(fill=tk.X, padx=50)
71          self.time_remaining_label.pack(fill=tk.X ,pady=15)
72          self.pause_button.pack(fill=tk.X, padx=50)
73
74          self.protocol("WM_DELETE_WINDOW", self.safe_destroy)
75
76      def setup_worker(self):
77          now = datetime.datetime.now()
78          in_25_mins = now + datetime.timedelta(minutes=25)
79          #in_25_mins = now + datetime.timedelta(seconds=3)
80          worker = CountingThread(self, now, in_25_mins)
81          self.worker = worker
82
83      def start(self):
84          if not hasattr(self, "worker"):
85              self.setup_worker()
86
87          self.task_name_entry.configure(state="disabled")
88          self.start_button.configure(text="Finish", command=self.finish_early)
89          self.time_remaining_var.set("25:00")
90          self.pause_button.configure(state="normal")
91          self.worker.start()
92
93      def pause(self):
94          self.worker.paused = not self.worker.paused
95          if self.worker.paused:
96              self.pause_button.configure(text="Resume")
97              self.worker.start_time = datetime.datetime.now()
98          else:
99              self.pause_button.configure(text="Pause")
100             end_timedelta = datetime.datetime.now() - self.worker.start_time
101             self.worker.end_time = self.worker.end_time + datetime.timedelta(seconds=
                    end_timedelta.seconds)
102
103     def finish_early(self):
104         self.start_button.configure(text="Start", command=self.start)
105         self.worker.end_now = True
106
107     def finish(self):
108         self.task_name_entry.configure(state="normal")
109         self.time_remaining_var.set("25:00")
110         self.pause_button.configure(text="Pause", state="disabled")
111         self.start_button.configure(text="Start", command=self.start)
112         del self.worker
113         msg.showinfo("Pomodoro Finished!", "Task completed, take a break!")
114
115     def update_time_remaining(self, time_string):
116         self.time_remaining_var.set(time_string)
117         self.update_idletasks()
118
119     def safe_destroy(self):
120         if hasattr(self, "worker"):
121             self.worker.force_quit = True
122             self.after(100, self.safe_destroy)
123         else:
124             self.destroy()
125
126 if __name__ == "__main__":
127     timer = Timer()
128     timer.mainloop()
```

Listing 7.1: A 25 Minute Timer

7.1.1 Timer

__init__

Everything in `__init__` should look familiar now. We create a `Frame` which holds all of our content. We have a `Label` which tells the user what to put in the `Entry`, a start `Button`, another `Label` holding the time remaining, and a pause `Button`. Within the pomodoro technique tasks aren't actually supposed to be paused, but life happens, so it may come in handy. Note that the pause `Button` is disabled by default, since we cannot pause a timer until it has begun.

We pack everything to fill the x direction giving us a single column layout. We use some padding to separate widgets vertically and to pull them off of the sides of the window. We then bind a method - `safe_destroy` - to the window close. This will be explained later.

setup_worker

Our "worker" is going to be a separate thread which will hold a reference to our `Timer` instance and call functions on it to update its widgets. Since a thread can only be run once, we cannot just set this up in our `__init__` and then call run each time we want to start a timer, we instead need to create a new instance each time. That's why we have this separate method.

To set up our `CountingThread` we need to give it a `start_time` and an `end_time`. As this method will only be run upon starting the timer, we can use `datetime.datetime.now()` to get the current time as our `start_time`. Since the pomodoro technique works in 25 minute blocks, we create our `end_time` by adding on a `datetime.timedelta(minutes=25)`. We create our `CountingThread` with these arguments and assign it to our `Timer` as `self.worker`.

start

If we don't have a worker, we will set one up. We then disable our `task_name_entry` and enable our `pause_button`, swap our `start_button` to a finish button, set the time `Label` to "25:00", and finally start off our worker.

pause

We use `not` to flip the paused attribute of our worker, allowing this function to work as both a pause and resume. If the worker is now paused we change the pause button to say "Resume" and set the current time as our worker's `start_time`. This will allow us to keep track of how long we were paused for an adjust the `end_time` accordingly.

On unpausing we set the button text back to "Pause" and calculate how long we were paused for by subtracting the `start_time` from the current time. This amount now needs to be added on to the worker's `end_time` to account for the time paused.

finish

Upon finishing we revert things back to their initial state, enabling our `task_name_entry`, disabling our `pause_button`, setting our clock back to "25:00", and changing our finish button back to a start button. We delete the reference to our worker as we no longer need it, since threads can only run once, before alerting the user that their time is up.

finish_early

If finishing early (by clicking the finish button which replaced our start button) We just need to swap the finish button back to a start button and set the `end_now` variable of our worker to `True`, which will set it up to handle the rest.

update_time_remaining

To update the timer on screen we simply call `set` on our `time_remaining_var` with the time returned from our `CountingThread`. We then call `update_idletasks` which forces the app to refresh its display. Without this the timer may occasionally appear to miss seconds.

safe_destroy

If the user was to start the timer and then close the window they would be left with a running thread still. In this case it seems as if the thread will throw an exception when it cannot reach the `Timer` instance and exit, but it is always best to ensure you do not leave an application with active threads still remaining. This ties up system resources and makes the user have to close them via some sort of task manager.

In our `safe_destroy` method we check to see if we have an assigned worker. If so this means the user has started the timer. We set the `force_quit` attribute of our worker to `True` which will cause it to `return` out of its `run` method and complete its duty. Before doing so it will `del` the reference in our `Timer` instance so that we know it has successfully ended. We use `self.after` to call this same method again every 100 milliseconds until the worker has removed the reference to itself from our `Timer`, in which case we are free to `destroy` the `Timer`.

Now let's have a look at exactly how our `CountingThread` works:

7.1.2 CountingThread

__init__ and run

Hopefully `__init__` is self explanitory, we are just setting up some variables. `master` will be our main window, `start_time` and `end_time` will be timestamps of when the pomodoro should start and end, and then we have 3 variables which keep track of whether or not the thread should continue running its loop.

`run` contains an infinite loop which first checks that none of our three variables which indicate that the loop should stop are true. If they aren't it will run its main loop to do some calculations and update the GUI. If the current time is past the set `end_time` we will signal to the `Timer` to `finish`.

If `end_now` is set, this means the user is finishing the task early, so this will jump to the finish method too. If `force_quit` is set then the user has closed the application window whilst the thread is still running, so we need to remove the thread from the main `Timer` before returning, which will end the thread.

The final `else continue` is hit when the `Timer` is paused, so the `CountingThread` needs to do nothing but still remain in its loop.

main_loop

In this method we need to find out the amount of time remaining and update the `Timer`'s clock appropriately. We grab the current time with `datetime.datetime.now()` and check if it's still less than our `end_time`. If it is we calculate the difference. We then use `divmod` to get the time in minutes and seconds which we can use with `.format` to create our next time string. We check once again for `force_quit` just to be sure before passing the time to `update_time_remaining`.

7.1.3 Next Iteration

Now that we have a basic timer application working we can build up some useful features to go along with it. Next iteration we will add a log screen to display finished tasks which have been stored in a sqlite database.

7.2 Keeping a Log

```
import sqlite3
import os
import functools
from tkinter import ttk

class CountingThread(threading.Thread):
    ...


class LogWindow(tk.Toplevel):
    def __init__(self, master):
        super().__init__()

        self.title("Log")
        self.geometry("600x300")

        self.notebook = ttk.Notebook(self)

        dates_sql = "SELECT DISTINCT date FROM pymodoros ORDER BY date DESC"
        dates = self.master.runQuery(dates_sql, None, True)

        for index, date in enumerate(dates):
            dates[index] = date[0].split()[0]

        dates = sorted(set(dates), reverse=True)

        for date in dates:
            tab = tk.Frame(self.notebook)

            columns = ("name", "finished", "time")

            tree = ttk.Treeview(tab, columns=columns, show="headings")

            tree.heading("name", text="Name")
            tree.heading("finished", text="Full 25 Minutes")
            tree.heading("time", text="Time")

            tree.column("name", anchor="center")
            tree.column("finished", anchor="center")
            tree.column("time", anchor="center")

            tasks_sql = "SELECT * FROM pymodoros WHERE date LIKE ?"
            date_like = date + "%"
            data = (date_like,)

            tasks = self.master.runQuery(tasks_sql, data, True)

            for task_name, task_finished, task_date in tasks:
                task_finished_text = "Yes" if task_finished else "No"
                task_time = task_date.split()[1]
                task_time_pieces = task_time.split(":")
                task_time_pretty = "{}:{}".format(task_time_pieces[0], task_time_pieces
                    [1])
                tree.insert("", tk.END, values=(task_name, task_finished_text,
                    task_time_pretty))

            tree.pack(fill=tk.BOTH, expand=1)

            self.notebook.add(tab, text=date)

        self.notebook.pack(fill=tk.BOTH, expand=1)


class Timer(tk.Tk):
```

```
63      def __init__(self):
64          ...
65
66          self.menubar = tk.Menu(self, bg="lightgrey", fg="black")
67
68          self.log_menu = tk.Menu(self.menubar, tearoff=0, bg="lightgrey", fg="black")
69          self.log_menu.add_command(label="View Log", command=self.show_log_window,
                accelerator="Ctrl+L")
70
71          self.menubar.add_cascade(label="Log", menu=self.log_menu)
72          self.configure(menu=self.menubar)
73
74          ...
75
76          self.bind("<Control-l>", self.show_log_window)
77
78          ...
79
80      def setup_worker(self):
81          ...
82
83      def start(self):
84          if not self.task_name_entry.get():
85              msg.showerror("No Task", "Please enter a task name")
86              return
87
88          ...
89          self.task_finished_early = False
90          ...
91
92      def pause(self):
93          ...
94
95      def finish_early(self):
96          self.start_button.configure(text="Start", command=self.start)
97          self.task_finished_early = True
98          self.worker.end_now = True
99
100     def finish(self):
101         ...
102         if not self.task_finished_early:
103             self.mark_finished_task()
104         del self.worker
105         msg.showinfo("Pomodoro Finished!", "Task completed, take a break!")
106
107     def update_time_remaining(self, time_string):
108         ...
109
110     def add_new_task(self):
111         task_name = self.task_name_entry.get()
112         self.task_started_time = datetime.datetime.now()
113         add_task_sql = "INSERT INTO pymodoros VALUES (?, 0, ?)"
114         self.runQuery(add_task_sql, (task_name, self.task_started_time))
115
116     def mark_finished_task(self):
117         task_name = self.task_name_entry.get()
118         add_task_sql = "UPDATE pymodoros SET finished = ? WHERE task = ? and date = ?"
119         self.runQuery(add_task_sql, ("1", task_name, self.task_started_time))
120
121     def show_log_window(self, event=None):
122         LogWindow(self)
123
124     def safe_destroy(self):
125         ...
126
127     @staticmethod
```

```
128     def runQuery(sql, data=None, receive=False):
129         conn = sqlite3.connect("pymodoro.db")
130         cursor = conn.cursor()
131         if data:
132             cursor.execute(sql, data)
133         else:
134             cursor.execute(sql)
135 
136         if receive:
137             return cursor.fetchall()
138         else:
139             conn.commit()
140 
141         conn.close()
142 
143     @staticmethod
144     def firstTimeDB():
145         create_tables = "CREATE TABLE pymodoros (task text, finished integer, date text)"
146         Timer.runQuery(create_tables)
147 
148 
149 if __name__ == "__main__":
150     timer = Timer()
151 
152     if not os.path.isfile("pymodoro.db"):
153         timer.firstTimeDB()
154 
155     timer.mainloop()
```

Listing 7.2: A Timer With a Log

7.2.1 Timer

There should be some nostalgia when working through this chapter, as a lot of code has been taken from Chapter 2. Most notably: `runQuery` and `firstTimeDb`.

When setting up our `Timer` instance we now have a `Menu` with a button to open the log. This is also bound to Control-L.

Upon starting a task, if there's no task name in our `task_name_entry` we will inform the user with a messagebox. You may have noticed that the `task_name_entry` was kind of pointless in the previous iteration, but now we have a database connected we will need the ability to name each task. We also have a boolean `task_finished_early` which will be used to mark whether or not a task was executed for the full 25 minutes. Within our `finish_early` method we will set this to `True` which affects whether or not the record is updated when we get to `finish`.

When we first `start` a task we add an entry into the database with the task's name and the date/time it started (via `add_new_task`). It is initially marked as not being worked on for the full 25 minutes. Once we hit the `finish` method we will update the value of the `finished` column if the task was not finished early (with `mark_finished_task`).

When creating and running our `Timer` instance, we will call `firstTimeDb` if the database file does not exist in the same directory as the app. This is the same as we did in Chapter 2 for our Todo list.

7.2.2 LogWindow

The `LogWindow` consists of two widgets from the ttk set: a `Notebook`, which we met in Chapter 3, and a `Treeview`. The `Notebook` is used to create a tabbed interface inside the window, and the `Treeview` will organise our information into a neat little table. This saves us from having to manually lay the information out using `Labels`.

We query our database for a list of dates then `enumerate` over them to replace each full `datetime` with just the date part. We need to use `date[0]` for each record as sqlite returns even single items in a `tuple`. We then use the somewhat strange looking `dates = sorted(set(dates), reverse=True)` to get a list of unique dates in descending order. We first cast the `list` to a `set` in order to remove duplicates, then `sorted` with `reverse=True` to order them descending. That way today's items are always first.

We once again loop over our now-ordered dates and create a new `Frame`, which will function as a tab in our `Notebook`, for each date. The tuple of strings will function as identifiers for each column and the `show="headings"` removes the default "icon" column from the `Treeview`. Without this we would get a blank first column. We use three calls to `.heading` to configure each column's heading, followed by three calls to `.column` to center-align our data.

Another query is run against our database to get all of the tasks which match the current date. We iterate over the results formatting the data in a friendlier way, and getting the times rather than the dates (since the date is written on the tab) before using `insert` to add the information into our `Treeview`. The blank string as the first argument tells the `Treeview` that this record has no parent, and the `tk.END` tells it to insert each record after all others. We then `pack` our `Treeview` into the tab and add the tab to our `Notebook`.

Once this has been done for each date, we finish off by packing our `Notebook`. With that our `LogWindow` is complete. Give it a go by running a couple of tasks then pressing Control-L to pop open the log.

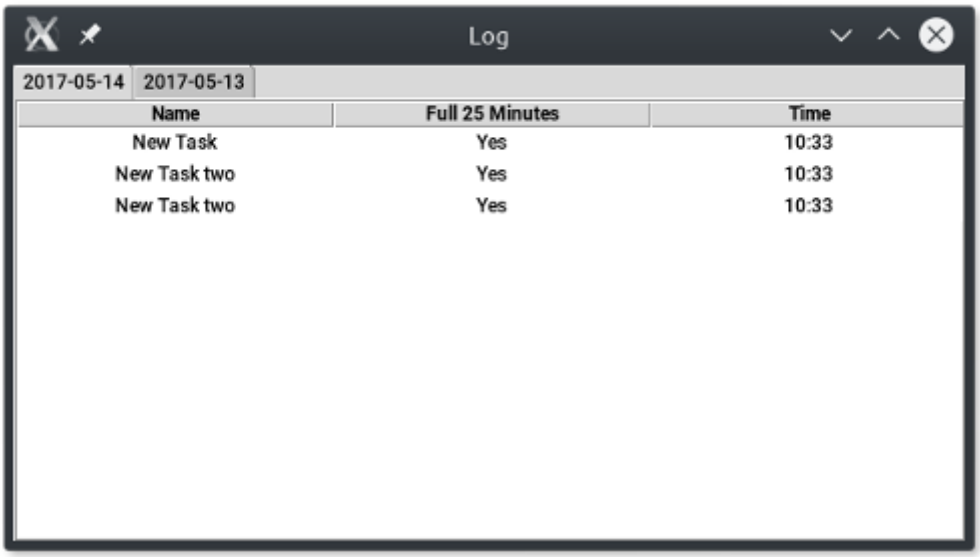

Figure 7.2: Our Log Window

7.2.3 Next Iteration

We'll finish up our timer by styling the `Treeview` using ttk's `Style` objects, as well as neatening up the main window by replacing some tk widgets with the ttk equivalent. We'll also add delete functionality via the log.

7.3 Our Finished Timer

```
...

class CountingThread(threading.Thread):
    ...


class LogWindow(tk.Toplevel):
    def __init__(self, master):
        ...
        self.tab_trees = {}

        style = ttk.Style()
        style.configure("Treeview", font=(None,12))
        style.configure("Treeview.Heading", font=(None, 14))

        dates = self.master.get_unique_dates()

        for index, date in enumerate(dates):
            dates[index] = date[0].split()[0]

        dates = sorted(set(dates), reverse=True)

        for date in dates:
            ...

            tree.pack(fill=tk.BOTH, expand=1)
            tree.bind("<Double-Button-1>", self.confirm_delete)
            self.tab_trees[date] = tree

            self.notebook.add(tab, text=date)

        self.notebook.pack(fill=tk.BOTH, expand=1)

    def confirm_delete(self, event=None):
        current_tab = self.notebook.tab(self.notebook.select(), "text")
        tree = self.tab_trees[current_tab]
        selected_item_id = tree.selection()
        selected_item = tree.item(selected_item_id)

        if msg.askyesno("Delete Item?", "Delete " + selected_item["values"][0] + "?",
            parent=self):
            task_name = selected_item["values"][0]
            task_time = selected_item["values"][2]
            task_date = " ".join([current_tab, task_time])
            self.master.delete_task(task_name, task_date)
            tree.delete(selected_item_id)

class Timer(tk.Tk):
    def __init__(self):
        ...

        style = ttk.Style()
        style.configure("TLabel", foreground="black", background="lightgrey", font=(None,
             16), anchor="center")
        style.configure("B.TLabel", font=(None, 40))
        style.configure("B.TButton", foreground="black", background="lightgrey", font=(
            None, 16), anchor="center")
        style.configure("TEntry", foregound="black", background="white")

        ...

        self.task_name_label = ttk.Label(self.main_frame, text="Task Name:")
        self.task_name_entry = ttk.Entry(self.main_frame, font=(None, 16))
```

```
61          self.start_button = ttk.Button(self.main_frame, text="Start", command=self.start,
                style="B.TButton")
62          self.time_remaining_var = tk.StringVar(self.main_frame)
63          self.time_remaining_var.set("25:00")
64          self.time_remaining_label = ttk.Label(self.main_frame, textvar=self.
                time_remaining_var, style="B.TLabel")
65          self.pause_button = ttk.Button(self.main_frame, text="Pause", command=self.pause,
                state="disabled", style="B.TButton")
66
67          ...
68
69          self.task_name_entry.focus_set()
70
71      def setup_worker(self):
72          ...
73
74      def start(self):
75          if not self.task_name_entry.get():
76              ...
77
78          if self.task_is_duplicate():
79              msg.showerror("Task Duplicate", "Please enter a different task name")
80              return
81
82          ...
83
84      def pause(self):
85          ...
86
87      def finish_early(self):
88          ...
89
90      def finish(self):
91         ...
92
93      def update_time_remaining(self, time_string):
94          ...
95
96      def add_new_task(self):
97          ...
98
99      def mark_finished_task(self):
100         ...
101
102     def show_log_window(self, event=None):
103         ...
104
105     def safe_destroy(self):
106         ...
107
108     def get_unique_dates(self):
109         dates_sql = "SELECT DISTINCT date FROM pymodoros ORDER BY date DESC"
110         dates = self.runQuery(dates_sql, None, True)
111
112         return dates
113
114     def get_tasks_by_date(self, date):
115         tasks_sql = "SELECT * FROM pymodoros WHERE date LIKE ?"
116         date_like = date + "%"
117         data = (date_like,)
118
119         tasks = self.runQuery(tasks_sql, data, True)
120
121         return tasks
122
123     def delete_task(self, task_name, task_date):
```

```
124         delete_task_sql = "DELETE FROM pymodoros WHERE task = ? AND date LIKE ?"
125         task_date_like = task_date + "%"
126         data = (task_name, task_date_like)
127         self.runQuery(delete_task_sql, data)
128 
129     def task_is_duplicate(self):
130         task_name = self.task_name_entry.get()
131         today = datetime.datetime.now().date()
132         task_exists_sql = "SELECT task FROM pymodoros WHERE task = ? AND date LIKE ?"
133         today_like = str(today) + "%"
134         data = (task_name, today_like)
135         tasks = self.runQuery(task_exists_sql, data, True)
136 
137         return len(tasks)
138 
139     @staticmethod
140     def runQuery(sql, data=None, receive=False):
141         ...
142 
143     @staticmethod
144     def firstTimeDB():
145         ...
146 
147 
148 if __name__ == "__main__":
149     ...
```

Listing 7.3: Our ttk Timer

7.3.1 Timer

__init__

Our widgets have now been swapped to their ttk counterparts and the styling options have been removed from their creation arguments. Ttk aims to keep declaration of widgets separate from their styling, meaning they will no longer support keyword arguments like `bg` when creating the instances. We instead create and use a `ttk.Style` object in order to adjust how our widgets look.

To achieve this we create a `Style` object and use its `configure` method to adjust style elements. Each ttk widget will have an associated class with which it gathers styling - usually a capital T followed by the object name, such as `TButton` or `TLabel` - but there are a couple of exceptions. The first argument to the `configure` method is the name of the style class we are changing and the following keyword arguments signify what we are changing.

When we configure `TLabel` in the first instance we are changing *all* `Labels` throughout our application. This is fine for us here as we only have two which both want the same colouring. We cannot do this for the `Button` class however as this affects the `Buttons` which appear in `messageboxes`.

In order to "subclass" a style we use a kind of dot-notation to specify inheritance. In our code you will see we define `B.TLabel`. This style inherits from the global `TLabel` we adjusted and allows us to build on top of it. In this case we want to inherit the colouring but increase the font size (the B stands for Big). Styling in this way prevents us from having to type `bg="lightgrey", fg="black"` for each widget.

We go on to define a Big `Button` styling with `B.TButton` and some global `Entry` styling with `TEntry`. Note that the font of an `Entry` cannot be set with the styling, so must be set upon creation as before.

To apply the non-global styles to our widgets we use the `style` keyword. Each one will default to the global (`TButton`, `TLabel`, etc) and if we want to specify an inherited style we pass the full style class as the argument. You can see this being done with our `Buttons` using "B.TButton" and our `time_remaining_label` using "B.TLabel".

We finish up our changes to `__init__` by setting focus to the `task_name_entry` when the user opens the app so that they don't have to click into it to begin typing.

Managing Tasks

All of the SQL has been moved from the `LogWindow` into the `Timer` for consistency. The two queries which should look familiar are `get_unique_dates` and `get_tasks_by_date`.

`delete_task` handles removing a task when it is double-clicked in the `LogWindow` (we will get to that soon).

`task_is_duplicate` is used to check whether we have a task with the same name on the current date. This is because we don't have a unique identifier for each task and we want to make sure we only delete one task at a time. If we had three tasks called "test" all done at the same time we would end up deleting them all when double clicking one of them in the log. We call this method from our `start` method and show a `messagebox` with an error if a task already exists.

7.3.2 LogWindow

Styling

The `Treeview` widget is one of the exceptions mentioned earlier when it comes to naming ttk `Styles`. Its class is just "Treeview" not "TTreeview". We use the `Style` to configure the font size of the items within our table. In order to change the font used in the headings we need to adjust the `Treeview.Heading` class. Again both of these `configures` apply globally to all `Treeviews` in our app.

Deleting

In order to get our delete functinality to work we need to bind double-click (`<Double-Button-1>` in tkinter) events to each `Treeview`. We also need to keep track of what `Treeviews` we have and which date they belong to. We do this using a dictionary called `tab_trees`. The key is the date and the item is the `Treeview` itself. Since our `Notebook` tabs are named after the dates this will allow us to access the relevant `Treeview` for the current tab.

Within `confirm_delete` we use the `tab` method of our `Notebook` to get the "text" attribute from our currently selected tab. This gives us the date of the tab currently being looked at. We use this date to fish out the relevant `Treeview` from `tab_trees` and grab the selected item's ID with `selection()`. We pass this ID to the `item` method in order to get a dictionary containing its information. If you want to see this dictionary add `print(selected_item)` after this line. The values of this item are stored within the "values" section of the dictionary.

We use an `askyesno messagebox` to confirm whether the user wants to delete this record. If so we get the task name and time from the "values", merge the date with the task time for specificity with delete statement, and then pass this information over to `delete_task` in our `Timer`. We finish off by calling the `delete` method of our tree to remove the item from the screen without having to re-build the whole page.

That's where we'll leave our pomodoro timer. We now have a 25 minute timer which contains a full log of all of our tasks, all handled automatically. We can also remove any tasks which we didn't want logged for any reason.

7.3.3 Further Development

If you'd like to continue work on this project as an exercise, try the following:

- Add scrolling to our log for those days when we are super productive.

- Add a way to re-order the tabs to be either ascending or descending.
- Add a to-do list to the app and have a way to select an item and have it auto-populate the task name entry.
- Allow the user to vary the timer length.

Chapter 8

Miscellaneous

That's it for all of the projects within this book. I hope you've learned enough to start developing your own GUI application with tkinter. I haven't covered absolutely everything in this book since I wanted all of the examples to be real, useful applications as opposed to small demonstrations of widgets. In this final chapter we'll just have a brief look at some things which I think will be useful to know but I didn't manage to cover in my examples.

8.1 Alternate Geometry Managers

8.1.1 Grid

Grid is a geometry manager with the same job as pack: to place your widgets into their parent. As you may have guessed from the name, `grid` treats your window as a literal grid and allows you to place widgets into a "cell" at a certain `row` and `column`. Their horizontal size is handled with `colspan` and the vertical size with `rowspan`. Widgets will expand via the use of a `sticky` argument which takes a combination of `"n"`, `"s"`, `"e"`, and `"w"` (north, south, east, west). This will make it stick to the particular end of its cell, so a `sticky` of "we" means the widget will stretch horizontally within its assigned cell. Widgets default to the center of their cell if there is no `sticky` value set.

We can `grid` widgets in any order we like, providing we specify their values correctly, since each one is assigned to a specific cell (or group of cells). With `pack` the order in which we `pack` our widgets defines their position. For example, when we are `packing` two `Buttons` with `side=tk.BOTTOM`, the first Button which is `packed` will appear at the very bottom, with the second above it. When adding more `Buttons` to the bottom of this window, we must ensure we `pack` them after the first one if we want to keep it at the bottom, whereas with `grid` we can just specify a smaller `row` value, and then `grid` it whenever we like.

The other main advantage of `grid` is that we don't have to use `Frames` if we wish to specify two `sides`. For example, take our find window from the text editor in chapter 6. In order to place our `Buttons` both at the bottom of the window and side-by-side we had to use a `Frame` `packed` to the bottom, then `pack` each widget to the left. If using `grid` we wouldn't need the extra `Frame`, we could simply give all of the `Buttons` the same `row`.

The reason I don't tend to use grid is simply because I find it unflexible when developing iteratively. If we accidentally `grid` a widget in the same `row` and `column` as another it will just overtake that cell, hiding the first widget. This means each time we want to add something we would potentially have to adjust the `row` and `column` of multiple other widgets.

I also find `pack` to be typically more readable than `grid`. Instead of having to compare numbers across multiple widgets to get a mental picture of what goes where, we have words like "bottom" and "left" right there in the code.

Despite my opinions, grid is a powerful tool, so if you feel it is better for the job than pack then I encourage you to use it. For some great examples with pictures check out the tkinterbook page over at effbot.org/tkinterbook/grid.htm.

8.1.2 Place

If you want to specify exact coordinates within the window to put something, place will do that for you. It's generally a pain to lay a window out with specifics, and there's much less room for the widgets to adapt with the window size, so place sees very little use.

To put a widget at (100, 300) within a window, use widget.place(x=100, y=300). Alternatively, you can use relx and rely to place a widget relative to its parent. relx=0.5, rely=0.5, anchor=tk.CENTER will keep a widget completely central in its parent.

placed widgets will overlap anything underneath them. This can be good or bad depending on your intentions.

8.2 Tk Widgets

There are still some widgets which I didn't manage to fit into any of the example apps. We'll have a brief overview of them here:

8.2.1 Checkbutton

A Checkbutton is essentially a checkbox with an attached label. The label is set with the text argument much like the other tkinter widgets. We can query whether or not the box has been checked by attaching a tkinter variable to it (StringVar, IntVar etc) with variable=self.my_variable. By deault the value of this variable will be 1 when checked and 0 when not. We can change this with the onvalue and offvalue arguments. Changing the linked variable directly will update the associated Checkbutton automatically.

Much like a normal Button, a Checkbutton can take a command argument to call a function whenever it is pressed.

8.2.2 Radiobutton

Somewhat similar to a Checkbutton, a Radiobutton is used to represent one choice out of a group of possible options. To group Radiobuttons, point them all to the same tkinter variable using the variable keyword. Each Radiobutton can then have its own unique value assigned with the value keyword, which becomes the value of the linked variable when this Radiobutton is selected.

Once again, the text argument will put a label beside the button. We can also bind a function via command.

By default a Radiobutton will look like it does on a standard HTML page (circular icon next to text with a dot inside the selected option). If you wish instead to have each option look like a regular button with the chosen option pressed in, setting the indicatoron argument to false will do this.

8.2.3 Checkbuttons and Radiobuttons in a Menu

A Menu can take contain Checkbuttons and Radiobuttons as well as the normal Buttons we used in our projects. These are added with .add_checkbutton(label="check", variable=self.checked) and .add_radiobutton(label="radio", variable=self.radio). The buttons will be linked to the supplied tkinter variable just like regular Checkbuttons and Radiobuttons.

8.2.4 OptionMenu

An `OptionMenu` is much like an HTML dropdown box. Unlike other `tkinter` widgets the `OptionMenu` doesn't rely on keyword arguments when creating an instance. Instead, instances are created like this: `om = OptionMenu(parent, variable, "option1", "option2", "option3")`. In this case `parent` is your root window, `variable` is a `tkinter` variable, and all of the following arguments are the options to choose from in the box.

If developing for Windows or OSX I would recommend using the `ttk` version of `OptionMenu` (and any ttk-supported widget to be honest), since it looks so much nicer. One thing to note with this version is the third argument will become the default. To clarify, we create an instance with `OptionMenu(parent, variable, "default choice", "choice 1", "choice2")`. The default choice will *not* appear in the list of available options unless re-declared as the 4th or higher argument, eg `(parent, variable, "medium", "low", "medium", "high")`.

A nicer way to specify the potential choices is to create a `tuple` and then unpack it when creating the `OptionMenu`, eg `(parent, variable, *choices)`.

8.3 Ttk Widgets

8.3.1 Combobox

A `Combobox` is a combination of an `Entry` and an `OptionMenu`. The user can either pick an option from the dropdown list or type their own. This is sometimes called a "select2" in the web development world. Unfortunately, typing in the `Entry` does not filter the values in the dropdown by default, so if that is your intention you will need to implement this manually. This can either be done by binding to the `<KeyRelease>` event, or by using the `postcommand` argument to bind a function which will run when the user clicks the dropdown arrow.

A `Combobox` can be instantiated by passing the parent as the first argument followed by the values as a sequence of strings. For example: `Combobox(parent, values=("one", "two", "three"))`. This widget can also be bound to a `StringVar` with the `textvariable` argument.

8.3.2 Progressbar

When running something which may take a long time we can use a `Progressbar` to let the user know that the application has not crashed.

If you have a quantifiable end goal, such as a number of open files to process, you can use a `determinate Progressbar` to show exactly how far through the process your application currently is. Determinate is the default mode of the `Progressbar` widget. Let's say you had a big list of open files to process - you would show the progress like so: `pb = Progressbar(parent, maximum=len(files))`. You now have a `Progressbar` with step count equal to the length of your file list. After processing each file, you can call `pb.step()` to increment progress by one. Once the `Progressbar` has reached its `maximum` it will return to empty, so you should `destroy` it (or its parent if it has a separate window).

If you have no idea how much work there is to do but still want to signal to the user that the app is processing, there is the `mode="indeterminate"` argument. This will create one of those animations where a small block bounces left and right until processing is complete. To begin this animation call `pb.start()`, and use `pb.stop()` when processing is complete (or use `destroy()` as before).

The length of a `Progressbar` can be set with the `length` argument, and for some reason you can also set it to vertical with `orient=tk.VERTICAL`.

8.4 Final Words

With that, we have come to the end of this book. Thanks very much for reading. I would love to hear your thoughts on this book - you can find me @Dvlv292 on twitter or Dvlv on reddit. Any comments, questions, or suggestions on the source code can be handled through Github. I am more than happy to alter the code and this book in order to improve it for people new to tkinter. As always in programming - nothing is ever finished!