Teek Documentation

Akuli

Jul 22, 2020

Contents

Teek is a pythonic way to write Tk GUIs in Python. See the README for an introduction to teek, and then click the tutorial link below to get started.

CHAPTER 1

Tutorials and Guides

1.1 Teek Tutorial

If you have never written any teek code before, you came to the right place. **You don't need to have any tkinter experience** in order to learn teek, although learning teek will be easier if you have used tkinter in the past.

Note: If you have **any trouble at all** following this tutorial, let me know and I'll try to make this tutorial better. You can contact me for other things as well, but I'm especially interested in making this tutorial beginner-friendly.

1.1.1 Tkinter, _tkinter, Tcl, Tk and teek

You need to install tkinter in order to install teek. That's correct, teek is an alternative to tkinter that needs tkinter. This section explains why.

When we say tkinter, we really mean a combination of two things that cannot be installed separately:

- The `_tkinter` module that lets you run a Tcl interpreter within Python. **Tcl** is a programming language that can be used without `_tkinter`.
- The pure-Python `tkinter` module uses `_tkinter`. When you do something like `root = tkinter.Tk()`, `tkinter` starts a Tcl interpreter using `_tkinter`. Tkinter widgets are just classes written in pure Python that run commands in the Tcl interpreter; to be more precise, they use a Tcl library called **Tk**.

Teek is an alternative to the pure-Python part; it uses a Tcl interpreter with `_tkinter`, just like `tkinter`, but it's much nicer to use in several ways as shown in the README.

---

tl;dr

Tcl is a programming language with a nice GUI library named **Tk**. **Tkinter** and **Teek** both use Tcl and Tk with `_tkinter`.

1.1.2 Installing Teek

Install Python 3.4 or newer and tkinter (yes, you need to have tkinter installed, there are more details about this above). Then run this:

```
python3 -m pip install --user teek
```

Use `py` instead of `python3` if you are on Windows.

1.1.3 Hello World!

```
import teek

window = teek.Window("Hello World")
label = teek.Label(window, "Hello World!")
label.pack()

window.on_delete_window.connect(teek.quit)
teek.run()
```

Run the program. It displays a tiny Hello World greeting.

Let's go through the program line by line.

```
import teek
```

This imports teek. If you get an `ImportError` or a `ModuleNotFoundError` from this step, make sure that you installed teek as explained above.

You can also `import teek as tk` and use `tk.Label` instead of `teek.Label`, but then people reading your code will see `tk.Label` there and think that it's probably a tkinter label, because it's quite common to `import tkinter as tk`. `teek.Label` is not much more work to type than `tk.Label` anyway (unlike `tkinter.Label`), so I recommend just `import teek`.

Don't do `from teek import *` because that confuses both tools that process code automatically, and people who read your code. `teek.Label(parent_widget, "hello")` obviously creates a teek label, but `Label(parent_widget, "hello")` creates a label. What is a label? If you have many star imports...

```
# this code is BAD!! DONT DO THIS!! NO!!!
from teek import *
from os import *
from sys import *
```

...and you need to find out where `Label` comes from, you have many things that it might be coming from; if `os` or `sys` had something called `Label`, it would replace teek's `Label`. Everyone reading this code need to know that neither `os` nor `sys` has anything named `Label`, which is bad. Also, if Python developers decide to add something called `Label` to `os` or `sys`, your code will break.

```
window = teek.Window("Hello World")
```

This creates a *`Window`* widget with title `"Hello World"`. A widget is an element of the GUI.

```
label = teek.Label(window, "Hello World!")
```

Many widgets need to go into another widget. *`Label`* is a widget that displays text, and this line of code puts it in our `window`. The widget that the label goes in is called the **parent** or the **master** widget of the label. Similarly, the label is said to be a **child** or **slave** of the window.

```
label.pack()
```

If you create a label into the window, it isn't displayed automatically. This line of code displays it.

Creating a child widget and displaying it in the parent are two separate things because this way you can choose how the widget shows up. There's more information about this *below*.

```
window.on_delete_window.connect(teek.quit)
```

This line tells teek to run *`teek.quit()`* when the window is closed. By default, nothing happens when the user tries to close the window. You can connect it to any other function or method as well, which is useful for things like "Do you want to save your changes" dialogs.

```
teek.run()
```

The code before this runs for just a fraction of a second, but this line of code stays running until we close the window. That's usually something between a few seconds and a few hours.

Note that instead of this...

```
label = teek.Label(window, "Hello World")
label.pack()
```

...we can also do this...

```
teek.Label(window, "Hello World").pack()
```

...because we create the variable once, and only use it once. However, this doesn't work:

```
label = teek.Label(window, "Hello World").pack()     # WRONG! common mistake
```

Look carefully: this does *not* set the `label` variable to a label; it sets that variable to what ever `the_actual_label_widget.pack()` returns, which is *not* same as the label widget itself. If you need to do more than one thing to a widget, set that widget to a variable and do all the things to that variable.

1.1.4 Options

Widget options can be used to change how widgets look and behave. For example, the text of a label is in an option named `text`.

```
>>> window = teek.Window()
>>> label = teek.Label(window, "blah blah")
>>> label.config['text']
'blah blah'
```

The only way to check the value of an option is `label.config['text']`, but you can set values of options in several ways:

- You can change the text after creating the label like `label.config['text'] = "new text"`. The label will display the new text automatically.
- When creating the label, you can pass options to it like `teek.Label(window, text="blah blah")`. Some common options can also be used without passing the option name explicitly with `text=`, e.g. `teek.Label(window, "blah blah")`. This is widget-specific, and it's documented in teek's documentation; for example, this label thing is documented in *`Label`* docs.

Sometimes the name of a widget option happens to be a reserved keyword in Python. For example, `in` is not a valid Python variable name because it's used in things like `'hello' in 'hello world'`:

```
>>> in = 'lol'
Traceback (most recent call last):
  ...
SyntaxError: invalid syntax
>>> label.pack(in=window)
Traceback (most recent call last):
  ...
SyntaxError: invalid syntax
```

To avoid this problem, you can use `in_` instead of `in`, and teek will handle it correctly:

```
>>> in_ = 'lol'
>>> in_
'lol'
>>> label.pack(in_=window)
```

Teek strips the last _ before it does anything with the option.

1.1.5 Tkinter Instructions

Tkinter is very popular, so if you want to know how to do something in tkinter, you can just google it. For example, if you want to change the text of a label after creating it, google "tkinter change label text" and you'll find a stackoverflow answer that does `some_label['text'] = 'new text'` and `some_label.config(text='new text')`. Neither of those works in teek, but both of them give errors with good messages that tell you what you need to do instead.

Sometimes teek and tkinter differ a lot more, and teek can't detect too tkintery ways to do things and give you particularly good errors. In these cases, use *teek's tkinter porting guide*.

1.1.6 Manual Pages

Note: This section assumes that you know the Tcl stuff explained *above*.

Sometimes stackoverflow answers don't contain the best possible solution because they are written by noobs who don't actually know Tk and tkinter very well. I see this quite often. Some of the people who answer tkinter questions on stackoverflow have 20+ years of *Tk* experience, but most answerers don't.

If you don't want to rely on stackoverflow or you want to do things like experienced Tk programmers do things, you should read Tk's manual pages. They are written for Tcl users and Tcl's syntax is quite different from Python syntax, so you will probably be somewhat confused at first. For example, let's say that you don't know how to change the text of a label after creating it. Figure it out like this:

1. Go to teek's label documentation by clicking this `Label` link. This tutorial and rest of teek's documentation are full of these links. Click them.
2. The `Label` link doesn't say anything about changing the text afterwards, but it has a link to a manual page. Click it.
3. In the manual page, press Ctrl+F and search for "text". You'll find a widget option whose "Command-Line Name" is `-text`. The leading - is common in Tcl syntax, but we won't need it in teek. So all we really need to do is to change the `'text'` widget option as shown *above*. We found what we were looking for.

BTW

The manual page names are like ttk_label(3tk) or after(3tcl). GUI things have `3tk` manual pages, and things documented in `3tcl` manual pages can be also used in Tcl programs that don't have a GUI.

If you use Linux, you can also install the manual pages on your system and read them without a web browser. For example, this command installs them on ubuntu:

```
sudo apt install tcl8.6-doc tk8.6-doc
```

The `8.6` makes sure that you get newest manual pages available. After installing the manual pages, you can read them like this:

```
man ttk_label
```

You can close the manual page by pressing q like quit. If you want to search, Ctrl+F won't work, but instead you can type `/text` followed by enter to search for `text`. All matches will be highlighted, and you can press n like next to go to the next match.

1.1.7 Buttons and callback functions

This code displays a button. Clicking the button runs the `on_click()` function.

```
import teek

def on_click():
    print("You clicked me!")

window = teek.Window("Button Example")
button = teek.Button(window, "Click me", command=on_click)
button.pack()

window.on_delete_window.connect(teek.quit)
teek.run()
```

Most of the code isn't very different from our label example. Let's go through the things that are different.

```
def on_click():
    print("You clicked me!")
```

This defines a function. If you have never defined functions before, you should *definitely* learn that before continuing with this tutorial. It'll make everything a lot easier. I have written more about defining functions here.

```
button = teek.Button(window, "Click me", command=on_click)
```

`Button` takes a parent widget and a text, just like *`Label`*, but *`Button`* also takes a function that is called when the button is clicked. Read that sentence carefully: *`Button`* takes a **function**. This is a common mistake:

```
button = teek.Button(..., command=on_click())      # ummm... it doesn't work like this!
↪!
```

`command=on_click()` does not do what was intended here; `command=on_click()` calls the `on_click` function because it has `()` after `on_click`, and when `on_click` has been called, it creates the button and passes the return value of `on_click` to it. Be careful to pass the function itself without calling it.

BTW

Teek lets you omit the `command=` part when creating buttons if you put the button text before the command, so this...

```
button = teek.Button(window, "Click me", on_click)
```

...does the same thing as this:

```
button = teek.Button(window, "Click me", command=on_click)
```

Here is another common mistake:

```
import time

def on_click():
    print("Doing something...")
    time.sleep(5)
    print("Done")
```

Here `time.sleep(5)` waits for 5 seconds. If you click the button now, the GUI will be frozen for 5 seconds. The button will look like it's pressed down, and you can't even close the window! This is bad, and that's why button callbacks must not do anything that takes longer than a tiny fraction of a second. See *concurrency documentation* if you need a button callback that runs for a long time.

1.1.8 Multiple child widgets in same parent

It's possible to put several different widgets into the same parent window with *`pack()`*, like this:

```
import teek

window = teek.Window("Pack Example")
teek.Label(window, "One").pack()
teek.Label(window, "Two").pack()

window.on_delete_window.connect(teek.quit)
teek.run()
```

The "Two" label will show up below the "One" label. If you don't want that, you can also put the labels next to each other:

```
teek.Label(window, "One").pack(side='left')
teek.Label(window, "Two").pack(side='left')
```

That's correct, both of them have `side='left'`. This means that the first widget goes all the way to the left edge, and the second goes to the *right* of that, and so on, so the widgets get stacked to the left edge. The default is `side='top'`, and that's why the widgets ended up below each other.

If you need more complex layouts, you can create a *`Frame`* and pack it, and add more widgets *inside* that *`Frame`*, like this:

```
import teek

window = teek.Window("Pack Example")
```

(continues on next page)

(continued from previous page)

```
big_frame = teek.Frame(window)
big_frame.pack(fill='both', expand=True)
teek.Label(big_frame, text="Left").pack(side='left', fill='both', expand=True)
teek.Label(big_frame, text="Right").pack(side='left', fill='both', expand=True)

status_bar = teek.Label(window, "This is a status bar")
status_bar.pack(fill='x')

window.geometry(300, 200)
window.on_delete_window.connect(teek.quit)
teek.run()
```

This example uses plenty of things from pack(3tk), but you know *how to read manual pages*, so you can figure out how it all works. If that isn't true at all, keep reading, and I'll explain how some of the things in the example work.

```
big_frame.pack(fill='both', expand=True)
```

This packs the big frame so that it stretches if you resize the window, and it fills as much space as possible. If you don't want to learn everything about pack, learn at least this "idiom".

```
status_bar.pack(fill='x')
```

This makes the status bar fill all the space it has horizontally. Mathematicians like to call it the `x` direction. Use `fill='y'` to fill vertically, or `fill='both'` to fill in both `x` and `y` directions.

```
window.geometry(300, 200)
```

This call to *`geometry()`* resizes the window so that it's bigger by default, and you can see how the widgets got laid out without first making the window bigger.

1.1.9 What now?

You are ready for creating a project in teek! All parts of teek's documentation are listed at left, but here are the things you will most likely need next:

- *Widget Reference*
- *Geometry Managers: pack, grid, place*

1.2 Porting from Tkinter

If you have some tkinter code and you would like to switch to teek, you came to the right place. This page lists differences between tkinter and teek. You might also find this page interesting if you are an experienced tkinter programmer.

1.2.1 Before you begin

Make sure you have good test coverage. If you don't know what test coverage is, I hope your project is quite small. In that case, make sure that you know how every part and feature of the GUI is supposed to work and how to check if it works well. Running a linter like flake8 may also help, and understanding how the code works doesn't hurt either.

Currently these tkinter widgets are missing in teek:

- `tkinter.Listbox`
- `tkinter.Menubutton`, `tkinter.ttk.Menubutton`
- `tkinter.OptionMenu`
- `tkinter.PanedWindow`, `tkinter.ttk.Panedwindow`
- `tkinter.Radiobutton`, `tkinter.ttk.Radiobutton`
- `tkinter.Scale`, `tkinter.ttk.Scale`
- `tkinter.ttk.Sizegrip`
- `tkinter.Spinbox`
- `tkinter.ttk.Treeview`

If the project uses some of these, you can still use them with *Tcl calls*. However, that's kind of painful, so if the project uses these widgets a lot, it's probably best to use tkinter for now or ask me to add the widget to teek.

1.2.2 To Get Started

If you find a file that contains `from tkinter import *`, you should immediately fix that. It is very bad style. Change the import to `import teek`, and then replace things like `Label` with `teek.Label`.

If you find code like `import tkinter as tk`, you can just change `tkinter` to `teek`, but I prefer changing it to `import teek` and then replacing `tk` with `teek` everywhere. That way, if you see `teek.Label` somewhere in the code you know right away that it's not a tkinter label. If there is `import tkinter`, change it to `import teek` and replace `tkinter.something` with `teek.something` everywhere.

Sometimes you may see this:

```
import tkinter as tk
from tkinter import ttk
```

Or much worse:

```
from tkinter import *
from tkinter.ttk import *
```

In these cases, you should replace both `tkinter.something` and `tkinter.ttk.something` with `teek.something` and do the `import teek`. The `teek` module contains everything you need, including Ttk widgets and a few non-Ttk widgets.

After this, try it out. It probably doesn't work yet, so keep reading.

1.2.3 Widget Name Differences

All tkinter GUIs use a `tkinter.Tk` object. There is no `Tk` object in teek; instead, you should create a *`Window`* object and use that. Usually you should also use *`Window`* instead of a *`Toplevel`* as explained in *`Window`* documentation. You can create as many teek windows as you want, and they actually use toplevels under the hood.

If the tkinter code creates multiple `tkinter.Tk` instances, it is probably broken. Replace all of them with *`teek.Window`*.

If you have code that uses `tkinter.Message`, you should use a `Label` instead. I believe message widgets were a thing before labels could handle multiline text, but nowadays the text of labels can contain `\n` characters.

If you have code that uses `ttk.Labelframe`, use `LabelFrame` instead. Look carefully: `Labelframe` and `LabelFrame` are not the same. Non-ttk tkinter supports `LabelFrame` only, but for some reason, `tkinter/ttk.py` has `Labelframe` in addition to `LabelFrame`.

1.2.4 Quitting

In tkinter, destroying the root window destroys the whole GUI and usually the program terminates soon after that. In teek, destroying a window doesn't quit the GUI, so instead of `root.destroy()` you need *`teek.quit()`*.

Trying to close a teek window does nothing by default. If you want the whole program to end instead, do this:

```
window.on_delete_window.connect(teek.quit)
```

If you want to close only the window that the user is closing (which is the case for dialogs and other such things), do this:

```
window.on_delete_window.connect(window.destroy)
```

1.2.5 Constants

Tkinter has lots of constants like `tkinter.BOTH`, but their values are just similar strings:

```
>>> import tkinter
>>> tkinter.BOTH
'both'
```

This means that `some_widget.pack(fill=tkinter.BOTH)` does the same thing as `some_widget.pack(fill='both')`. Some programmers use constants like `tkinter.BOTH` while others prefer to just write `'both'`. I think these constants are dumb, which is why teek doesn't have them. Use strings like `'both'` in teek.

1.2.6 Run

Use *`teek.run()`* instead of tkinter's `root.mainloop()` or `tkinter.mainloop()`.

1.2.7 Options

Options are used differently in tkinter and teek. For example, `button['text']`, `button.cget('text')`, `button.config('text')[-1]` and `button.configure('text')[-1]` are all valid ways to get the text of a button. In teek, none of these work, and you instead do `button.config['text']`. However, teek raises good error messages:

```
>>> button = teek.Button(teek.Window(), "some text")
>>> button.cget('text')
Traceback (most recent call last):
    ...
TypeError: use widget.config['option'], not widget.cget('option')
>>> button['text']
Traceback (most recent call last):
    ...
TypeError: use widget.config['option'], not widget['option']
>>> button.config['text']
'some text'
```

1.2.8 Widget-specific Differences

Most widgets work more or less the same way in teek and tkinter, but not all widgets do. Some of the biggest differences are listed here, but not everything is; refer to *the documentation* of the widget that is causing errors for more details.

Button and CheckButton Tkinter buttons and checkbuttons have a `command` option that is set to a function that runs when the button is clicked, but that's a *`Callback`* object in teek:

```
>>> button.config['command'] = print
Traceback (most recent call last):
    ...
ValueError: cannot set the value of 'command', maybe use widget.config['command'].
↪connect() instead?
>>> button.config['command'].connect(print)
```

This way more than one callback can be easily connected to the button.

Text and Notebook Many things are very different (read: much better and more pythonic) in teek. You probably need to read most of teek's *text widget docs* or *notebook docs* anyway, so I won't even try to summarize everything here.

Entry Instead of `insert`, `delete` and `get` methods, there is a settable *`text`* attribute.

1.2.9 Dialogs

Dialog functions are named differently in teek. For example, instead of `filedialog.askopenfilename()` you use *`teek.dialog.open_file()`*. Unlike in tkinter, you don't need to import anything special in order to use the dialog functions; `import teek` is all you need, and after that, you can do `teek.dialog.open_file()`.

1.2.10 Binding

Teek's bind stuff is documented *here*. As you can see there, we have some differences to tkinter. First of all, if you want anything to work at all, you need to pass `event=True` to *`bind()`* to get tkinter-like event objects. However, this is a common thing to do in tkinter:

```
widget.bind('<SomeEvent>', lambda event: some_function())
```

Tkinter always gives an `event` argument to bind callbacks, and the lambda discards it because `some_function` must be called like `some_function()`, not `some_function(event)`. If you just pass `event=True`, you end up with code like this...

```
widget.bind('<SomeEvent>', (lambda event: some_function()), event=True)
```

...which can be simplified a lot because not using `event=True` does the same thing as the lambda:

```
widget.bind('<SomeEvent>', some_function)
```

If you do need the event object, watch out for differences in the attributes. For example, tkinter's `event.x_root` is `event.rootx` in teek. This is for consistency with *`event_generate()`*.

Note that tkinter's `bind` discards all old bindings, but this doesn't happen in teek. For example, if you do this...

```
widget.bind('<SomeEvent>', func1)
widget.bind('<SomeEvent>', func2)
```

...only `func2` is bound in tkinter, but both are bound in teek.

Tkinter's bind takes an `add=True` argument that tells it to not forget old bindings, and you can safely get rid of it. If you see some tkinter code that relies on the discarding behaviour, which I don't see very often, you need to use *`Widget.bindings`* to unbind the old function.

Speaking of unbinding, tkinter also has an `unbind()` method. It works like this when used correctly:

```
func_id = widget.bind('<SomeEvent>', func)
...
widget.unbind('<SomeEvent>', func_id)
```

Searching for `def unbind` in tkinter's source code reveals that `widget.unbind` actually discards all bindings of `<SomeEvent>`, and if the `func_id` is given, it also cleans things up. Teek does the cleanup automatically for you when the widget is destroyed (see *`destroy()`*).

1.2.11 Widget Methods

Tkinter's widgets have some methods that are available in all widgets, and they don't actually do anything with the widget. For example, `any_widget.after(1000, func)` runs `func()` in the *event loop* after waiting for 1 second. In teek, things that don't need a widget in order to work are functions, not widget methods. Here is a list of them:

Tkinter	Teek
`any_widget.after(milliseconds, cb)`	*`teek.after()`*
`any_widget.after_idle(cb)`	*`teek.after_idle()`*
`any_widget.update()`	*`teek.update()`*
`any_widget.tk.call()`	*`teek.tcl_call()`*
`any_widget.tk.eval()`	*`teek.tcl_eval()`*
`any_widget.tk.createcommand()`	*`teek.create_command()`*
`any_widget.tk.deletecommand()`	*`teek.delete_command()`*
`any_widget.mainloop()`	*`teek.run()`*
`root.destroy()`	*`teek.quit()`*

There are also some things that must be done with `any_widget.tk.call()` in tkinter, but teek has nicer support for them:

Tkinter	Teek
`any_widget.call('tk', 'windowingsystem')`	*`teek.windowingsystem()`*

1.2.12 Variable Objects

`DoubleVar` is *`FloatVar`* in teek because not all python users know that `double` means a precise `float` in programming languages like C. Other variable classes have same names.

There is no `trace()` method, but there is a *`write_trace`* attribute.

1.2.13 Font Objects

Tkinter has one font class, `tkinter.font.Font`, which represents a font that has a name in Tcl. There are two font classes in teek, and usually you should use *`NamedFont`* in teek when `tkinter.font.Font` is used in tkinter. See *font documentation* for details.

1.2.14 Tcl Calls

In tkinter, you might see code like this:

```
if root.tk.call('tk', 'windowingsystem') == 'aqua':
    ...some mac specific code...
```

Here `root.tk.call('tk', 'windowingsystem')` calls `tk windowingsystem` in Tcl, and that returns `'win32'`, `'aqua'` or `'x11'` as documented in tk(3tk). Notice that the return type is a string, but it's not specified anywhere. Teek is more explicit:

```
if tk.tcl_call(str, 'tk', 'windowingsystem') == 'aqua':
    ...
```

`1.2 == '1.2'` is false in python, but there is no distinction like that in Tcl; all objects are essentially strings, and `1.2` is literally the same thing as `'1.2'`. There is no good way to figure out what type tkinter's `root.tk.call` will return, and it's easiest to try it and see.

Teek gets rid of this problem by requiring explicit return types everywhere. If you want a Tcl call to return a string, you pass it `str`. See *Tcl Calls* for more documentation.

1.3 Concurrency

This page is all about running things concurrently in teek. That means doing something else while the GUI is running.

1.3.1 Threads

Here is some code that pastebins a Hello World to dpaste.com.

```
import requests
import teek


class Dpaster(teek.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.paste_button = teek.Button(self, "Pastebin a hello world", self.paste)
        self.paste_button.pack()

        self.url_label = teek.Label(self)
        self.url_label.pack()

    def paste(self):
        self.url_label.config['text'] = "Pasting..."

        # api docs: http://dpaste.com/api/v2/
        print("Starting to paste")
        response = requests.post('http://dpaste.com/api/v2/',
                                 data={'content': 'Hello World'})
        response.raise_for_status()
        url = response.text.strip()
        print("Pasted successfully:", url)
```

(continues on next page)

(continued from previous page)

```
        self.url_label.config['text'] = url


window = teek.Window("dpaster")
Dpaster(window).pack()
window.geometry(250, 100)
window.on_delete_window.connect(teek.quit)
teek.run()
```

Run the program. If you click the pasting button, the whole GUI freezes for a couple seconds and the button looks like it's pressed down, and when the pastebinning is done, everything is fine again. **The freezing is not nice.**

In this documentation, functions and methods that take a long time to complete are called **blocking**. Our `paste()` method is blocking, and using a blocking function or method as a button click callback freezes things.

If you have code like this...

```
import time


def one_thing():
    print("one thing")
    time.sleep(1)
    print("one thing")
    time.sleep(1)
    print("one thing")


def another_thing():
    print("another thing")
    time.sleep(1)
    print("another thing")
    time.sleep(1)
    print("another thing")


one_thing()
another_thing()
```

...Python obviously doesn't run `one_thing()` and `another_thing()` at the same time; it'll first run `one_thing()`, and when it's done, it'll run `another_thing()`. However, if we add `import threading` to the top of the program and change the last 2 lines to this...

```
threading.Thread(target=one_thing).start()
another_thing()
```

...the functions *will* run at the same time.

Note: We are doing `target=one_thing`, **not** `target=one_thing()`. The `()` at the end of `one_thing()` tell Python to run the function right away, but instead of that, we want to run it in the thread.

The good **ehm** **awesome** news is that threads work nicely in teek. Add `import threading` to the top of the file, and add this method to the `Dpaster` class...

```
def start_pasting(self):
    threading.Thread(target=self.paste).start()
```

... and change the line that creates `self.paste_button` to this:

```
self.paste_button = teek.Button(self, "Pastebin a hello world", self.start_pasting)
```

Let's try to run the program again. Clicking the button gives this error:

```
Traceback (most recent call last):
    ...
RuntimeError: init_threads() wasn't called
```

Let's fix it by adding `teek.init_threads()` before the line that creates `window`. All in all, the code looks like this now:

```
import threading

import requests
import teek


class Dpaster(teek.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.paste_button = teek.Button(self, "Pastebin a hello world", self.start_
↪pasting)
        self.paste_button.pack()

        self.url_label = teek.Label(self)
        self.url_label.pack()

    def start_pasting(self):
        threading.Thread(target=self.paste).start()

    def paste(self):
        self.url_label.config['text'] = "Pasting..."

        # api docs: http://dpaste.com/api/v2/
        print("Starting to paste")
        response = requests.post('http://dpaste.com/api/v2/',
                                 data={'content': 'Hello World'})
        response.raise_for_status()
        url = response.text.strip()
        print("Pasted successfully:", url)

        self.url_label.config['text'] = url


teek.init_threads()
window = teek.Window("dpaster")
Dpaster(window).pack()
window.geometry(250, 100)
window.on_delete_window.connect(teek.quit)
teek.run()
```

Run the program. It works!

How does it work, and why did it freeze??

Note: This section assumes that you know *event loop stuff*.

Button click callbacks are ran in the event loop. If the button command takes 2 seconds or so to run, the event loop will run it for 2 seconds, but it can't do anything else while it's doing that (see the event loop stuff link above to understand why). However, `start_pasting()` is not blocking, and running that from the event loop is fine.

If you have used `tkinter` before, the above program probably looks quite wrong to you, because in tkinter, running anything like `self.url_label.config['text'] = "Pasting..."` in a thread is **BAD**. Threads and tkinter don't work together well, and you must not do *any* tkinter stuff from threads. If you wanted to write the pastebinning program in tkinter, you would need to do quite a few things yourself:

- Create a `Queue` that will contain texts of `self.url_label`. Queues can be used from threads **and** from Tk's event loop, so we can use them to communicate between the pastebinning thread and the event loop.
- Create a method that gets a text from the queue and sets it to `self.url_label`.
  - This method needs to use the tkinter label, so it must not be called from a thread because tkinter and threads don't mix well. The only other way to call it is from tkinter's event loop.
  - Because the method is called from tkinter's event loop, it must not block; that is, it can't wait until a message arrives to the queue from the thread. If there are no messages in the queue, it must do nothing.
  - Because the method can't wait for messages in the thread, and it can only check if there are messages, it must be ran repeatedly e.g. 20 times per second. The thread will then add a message to the queue, and the queue clearing method will do something with that message soon.
  - Because the method must be called 20 times per second, you must tell the event loop to run it 20 times per second. The only way to do this is with *after callbacks*.

If you accidentally call a tkinter thing from a thread, you may get very weird behaviour (but in teek, you get a `RuntimeError` as shown above):

- Things may work 90% of the time and break 10% of the time.
- Everything may work just fine on your computer but not on someone else's computer.
- When things break, you get confusing error messages that don't necessarily say anything about threads.

Furthermore, beginners often want to use threads with tkinter, and they struggle with it a lot, which is no surprise. Threading with tkinter is hard.

Teek's `init_threads()` does the hard things for you:

`teek.`**`init_threads`**(*poll_interval_ms=50*)
Allow using teek from other threads than the main thread.

This is implemented with a queue. This function starts an *after callback* that checks for new messages in the queue every 50 milliseconds (that is, 20 times per second), and when another thread calls a teek function that does a *Tcl call*, the information required for making the Tcl call is put to the queue and the Tcl call is done by the after callback.

Note: After callbacks don't work without the event loop, so make sure to run the event loop with *`run()`* after calling *`init_threads()`*.

poll_interval_ms can be given to specify a different interval than 50 milliseconds.

When a Tcl call is done from another thread, that thread blocks until the after callback has handled it, which is slow. If this is a problem, there are two things you can do:

- Use a smaller poll_interval_ms. Watch your CPU usage though; if you make poll_interval_ms too small, you might get 100% CPU usage when your program is doing nothing.
- Try to rewrite the program so that it does less teek stuff in threads.

If you use init_threads(), you can also use this decorator:

teek.**make_thread_safe**(*func*)

A decorator that makes a function safe to be called from any thread.

Functions decorated with this always run in the event loop, and therefore in the main thread.

Most of the time you don't need to use this yourself; teek uses this a lot internally, so most teek things are already thread safe. However, if you have code like this...

```
def bad_func123():
    func1()
    func2()
    func3()
```

...where func1, func2 and func3 do teek things and you need to call func123 from a thread, it's best to decorate func123:

```
@teek.make_thread_safe
def good_func123():
    func1()
    func2()
    func3()
```

This may make func123 noticably faster. If a function decorated with make_thread_safe() is called from some other thread than the main thread, it needs to communicate between the main thread and teek's event loop, which is slow. However, with good_func123, there isn't much communication to do: the other thread needs to tell the main thread to run the function, and later the main thread tells the other thread that the function has finished running. The bad_func123 function does this 3 times, once in each line of code.

Note: Functions decorated with make_thread_safe() must not block because they are ran in the event loop. In other words, this code is bad, because it will freeze the GUI for about 5 seconds:

```
@teek.make_thread_safe
def do_stuff():
    time.sleep(5)
```

1.3.2 Letting the user know that something is happening

Here is a part of our example program above.

```
def __init__(self, *args, **kwargs):
    ...
    self.paste_button = teek.Button(self, "Pastebin a hello world", self.start_
↪pasting)
```

(continues on next page)

(continued from previous page)

```
    ...

def start_pasting(self):
    threading.Thread(target=self.paste).start()

def paste(self):
    self.url_label.config['text'] = "Pasting..."
    ...
    self.url_label.config['text'] = url
```

Can you see the problem? The paste button can be clicked while `paste()` is running in the thread. If the user does that, we have two pastes running at the same time in different threads. That's not nice.

A simple alternative is to make the button grayed out in the paste function:

```
def paste(self):
    self.paste_button.state.add('disabled')
    self.url_label.config['text'] = "Pasting..."
    ...
    self.url_label.config['text'] = url
    self.paste_button.state.remove('disabled')
```

See *`Widget.state`* for documentation about the state thing.

If you don't want to disable widgets or you would need to disable a widget and all widgets in it, you can use *`Widget.busy()`* instead, like this:

```
def paste(self):
    with self.busy():
        self.url_label.config['text'] = "Pasting..."
        ...
        self.url_label.config['text'] = url
```

Here is the reference.

`Widget.`**`busy_hold`**`()`
: See `tk busy hold` in busy(3tk).

  New in Tk 8.6.

`Widget.`**`busy_forget`**`()`
: See `tk busy forget` in busy(3tk).

  New in Tk 8.6.

`Widget.`**`busy_status`**`()`
: See `tk busy status` in busy(3tk).

  This Returns True or False.

  New in Tk 8.6.

`Widget.`**`busy`**`()`
: A context manager that calls *`busy_hold()`* and *`busy_forget()`*.

  Example:

  ```
  with window.busy():
      # window.busy_hold() has been called, do something
      ...
  ```

(continues on next page)

(continued from previous page)

```
# now window.busy_forget() has been called
```

For more advanced things, you can also use a separate *Progressbar* widget.

1.3.3 After Callbacks

Sometimes *threads* are overkill. Here is a clock program:

```
import threading
import time

import teek


class Clock(teek.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.label = teek.Label(self)
        self.label.pack()

        threading.Thread(target=self.updater_thread).start()

    def updater_thread(self):
        while True:
            self.label.config['text'] = time.asctime()
            time.sleep(1)


teek.init_threads()
window = teek.Window("Clock")
Clock(window).pack()
window.on_delete_window.connect(teek.quit)
teek.run()
```

BTW

If you close the window that the above program creates, you get a `RuntimeError` saying that *init_threads()* wasn't called, because closing the window calls *quit()* and *init_threads()* would need to be called again after a *quit()*. You can make the program exit cleanly by replacing this...

```
self.label.config['text'] = time.asctime()
```

...with this:

```
try:
    self.label.config['text'] = time.asctime()
except RuntimeError:
    # the program is quitting
    return
```

Returning from the thread target will stop the thread, and Python will exit because no more threads are running.

The repeatedly called `time.sleep(1)` in `updater_thread()` tells you that after callbacks might be a better alternative. They work like this:

```
import time

import teek


class Clock(teek.Frame):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        self.label = teek.Label(self)
        self.label.pack()
        self.updater_callback()

    def updater_callback(self):
        self.label.config['text'] = time.asctime()

        # tell tk to run this again after 1 second
        teek.after(1000, self.updater_callback)


window = teek.Window("Clock")
Clock(window).pack()
window.on_delete_window.connect(teek.quit)
teek.run()
```

`teek.after(1000, self.updater_callback)` runs `self.updater_callback()` in Tk's event loop after 1000 milliseconds; that is, 1 second.

`teek.`**`after`**(*ms*, *callback*, *args=()*, *kwargs=None*)
: Run `callback(*args, **kwargs)` after waiting for the given time.

  The *ms* argument should be a waiting time in milliseconds, and *kwargs* defaults to `{}`. This returns a timeout object with a `cancel()` method that takes no arguments; you can use that to cancel the timeout before it runs.

`teek.`**`after_idle`**(*callback*, *args=()*, *kwargs=None*)
: Like *`after()`*, but runs the timeout as soon as possible.

See also after(3tcl).

It's also possible to cancel a timeout before it runs. *`after()`* and *`after_idle()`* return **timeout objects**, which have a method for canceling:

`timeout_object.`**`cancel`**`()`
: Prevent this timeout from running as scheduled.

  `RuntimeError` is raised if the timeout has already ran or it has been cancelled.

Timeout objects also have a useful string representation for debugging:

```
>>> teek.after(1000, print)       # doctest: +ELLIPSIS
<pending 'print' timeout 'after#...'>
```

CHAPTER 2

Reference

Use this section when you want to know something about things that were not covered in the tutorial.

2.1 Geometry Managers: pack, grid, place

Geometry managers are ways to add widgets inside other widgets. For example, `pack` is a geometry manager. This page contains more information about pack and two other geometry managers.

2.1.1 Pack

This is a simple geometry manager useful for laying out big things. If you want to have two big widgets next to each other, this is the right choice.

See also:

There is a beginner-friendly introduction to pack *here*.

Pack options have a `-` in front of them in Tcl, but teek hides that, so Tcl code like `-anchor center` looks like `anchor='center'` in teek.

Teek widgets have these methods:

`teek.Widget.`**`pack`**`(`*`**kwargs`*`)`
: Search for `pack configure` in pack(3tk) for details. If you want to use the `-in` option, you can't do `widget1.pack(in=widget2)` because `in` is a reserved keyword in Python and you get a syntax error, but teek lets you do `widget1.pack(in_=widget2)` instead.

`teek.Widget.`**`pack_forget`**`()`
: See `pack forget` in pack(3tk).

`teek.Widget.`**`pack_info`**`()`
: This returns the keyword arguments passed to *`pack()`* with string keys. The types of values are as follows:

  - The value of `'in'` is a teek widget.

- The value of `'expand'` is True or False.
- The values of `'ipadx'` and `'ipady'` are *ScreenDistance* objects.
- The values of `'padx'` and `'pady'` are lists of *ScreenDistance* objects. Each list contains 1 or 2 items; see `-padx` in pack(3tk) for details.
- Other values are strings.

See also `pack info` in pack(3tk). The returned dictionary is a valid `kwargs` dict for *pack()*, so you can do this:

```
pack_info = widget.pack_info()
widget.pack_forget()
...do something...
widget.pack(**pack_info)
```

`teek.Widget.`**`pack_slaves`**`()`
Returns a list of other *Widgets* packed into the widget. See `pack slaves` in pack(3tk) for more details.

2.1.2 Grid

This geometry manager is useful for making griddy things that would be hard or impossible to do with pack. Here is an example:

```
import teek


window = teek.Window("Calculator")

rows = [
    ['7', '8', '9', '*', '/'],
    ['4', '5', '6', '+', '-'],
    ['1', '2', '3', None, None],
    [None, None, '.', None, None],
]

for row_number, row in enumerate(rows):
    for column_number, text in enumerate(row):
        if text is not None:
            button = teek.Button(window, text, width=3)
            button.grid(row=row_number, column=column_number, sticky='nswe')

zerobutton = teek.Button(window, '0')
zerobutton.grid(row=3, column=0, columnspan=2, sticky='nswe')
equalbutton = teek.Button(window, '=')
equalbutton.grid(row=2, column=3, rowspan=2, columnspan=2, sticky='nswe')

for row_or_column in (window.grid_rows + window.grid_columns):
    row_or_column.config['weight'] = 1

window.on_delete_window.connect(teek.quit)
teek.run()
```

Let's go through some of that line by line.

```
for row_number, row in enumerate(rows):
    for column_number, text in enumerate(row):
```

This is a way to loop over the row list with indexes. For example, if `text` is `'*'`, then `row_number` is 0 and `column_number` is 3, because `text` is the fourth element of the first sublist of `rows`.

```
button.grid(row=row_number, column=column_number, sticky='nswe')
```

The `sticky='nswe'` makes the button fill all the space it has in the grid cell. The `n` means "north" (up), `w` means "west" (left), etc.

```
zerobutton.grid(row=3, column=0, columnspan=2, sticky='nswe')
```

`columnspan=2` makes the button *span* two columns, so some of it is in one column, and rest of it is in the other. The default is `columnspan=1`.

```
for row_or_column in (window.grid_rows + window.grid_columns):
    row_or_column.config['weight'] = 1
```

This loops through all grid rows and columns of the widget, and makes everything stretch as the window is resized. Comment out these lines and resize the window to understand why I did this. See *Grid Row and Column Objects* below for details.

`teek.Widget.`**`grid`**`(**kwargs)`
: Very similar to *`pack()`*. See `grid configure` in grid(3tk) for details.

`teek.Widget.`**`grid_forget`**`()`
: See `grid forget` in grid(3tk).

`teek.Widget.`**`grid_info`**`()`
: Very similar to *`pack_info()`*. The types of values are as follows:

  - The value of `'in'` is a teek widget.
  - The values of `'ipadx'` and `'ipady'` are *`ScreenDistance`* objects.
  - The values of `'padx'` and `'pady'` are lists of *`ScreenDistance`* objects. Each list contains 1 or 2 items; see `-padx` in pack(3tk) for details.
  - The values of `'row'`, `'rowspan'`, `'column'` and `'columnspan'` are integers.
  - Other values are strings.

`teek.Widget.`**`grid_slaves`**`()`
: Similar to *`pack_slaves()`*. Use *`row_or_column.get_slaves()`* if you need the `-row` and `-column` options of `grid slaves` in grid(3tk).

`teek.Widget.`**`grid_rows`**
`teek.Widget.`**`grid_columns`**
: Lists of *row objects or column objects*.

Grid Row and Column Objects

Tk has some options and other things that can be done to rows or columns of a grid. These are represented with row objects and column objects in teek.

```
>>> window = teek.Window()
>>> window.grid_rows
[]
>>> teek.Label(window, "label text").grid()     # goes to row 0, column 0
>>> window.grid_rows
[<grid row 0: has a config attribute and a get_slaves() method>]
```

(continues on next page)

(continued from previous page)

```
>>> window.grid_columns
[<grid column 0: has a config attribute and a get_slaves() method>]
>>> window.grid_columns[0].config['weight']
0.0
>>> window.grid_columns[0].get_slaves()
[<teek.Label widget: text='label text'>]
```

Here is the reference:

`row_or_column.`**`config`**
: An object that represents row or column options. Similar to *`Widget.config`*.

  See `grid columnconfigure` and `grid rowconfigure` in grid(3tk) for the available options. `'weight'` is a float, `'minsize'` and `'pad'` are *`ScreenDistance`* objects and `'uniform'` is a string.

`row_or_column.`**`get_slaves`**`()`
: Returns a list of widgets in the row or column.

  This calls `grid slaves` documented in grid(3tk) with a `-row` or `-column` option.

2.1.3 Place

This geometry manager is useful for placing things relatively. For example, this puts a button right in the middle of its parent widget:

```
button.place(relx=0.5, rely=0.5, anchor='center')
```

Here `relx` and `rely` mean "relative x" and "relative y"; that is, values between 0 and 1. The `anchor='center'` says that the center of the *button* goes to the position given with `relx` and `rely`.

Place can be also used with absolute positions given in pixels:

```
# 100 pixels down and 200 right from top left corner of the parent widget
button.place(x=100, y=200)
```

Usually that's a **bad idea**. 100 pixels on your screen might look different than 100 pixels on someone else's screen if that other system happens to use bigger fonts or something else like that. However, placing with pixels can be useful when working with other things that use pixels. For example, if you *bind* some mouse stuff and the user clicks something, you get the mouse x and y coordinates in pixels.

`teek.Widget.`**`place`**`(**`*`kwargs`*`)`
: See `place configure` in place(3tk) for details. This is similar to *`pack()`*.

`teek.Widget.`**`place_forget`**`()`
: See `place forget` in place(3tk).

`teek.Widget.`**`place_info`**`()`
: Very similar to *`pack_info()`* and *`grid_info()`*. The types of values are as follows:

  - The value of `'in'` is a teek widget.
  - The values of `'width'`, `'height'`, `'x'` and `'y'` are *`ScreenDistance`* objects.
  - The values of `'relwidth'`, `'relheight'`, `'relx'` and `'rely'` are floats.
  - Other values are strings.

  See also `place info` in place(3tk).

`teek.Widget.place_slaves()`
Returns a list of other *`Widgets`* placed into the widget.

2.2 Widget Reference

This is a big list of widgets and explanations of what they do. As usual, don't try to read all of it, unless you're very bored; it's best to find what you're looking for, and ignore rest of this.

2.2.1 Canvas Widget

Manual page: canvas(3tk)

This widget is useful for displaying drawings and other things. See examples/paint.py for an example.

Note: This widget is **not** a Ttk widget because there is no Ttk canvas widget. However, that's not a problem because usually canvas widgets contain things that shouldn't be colored based on the current Ttk theme anyway.

Hello World!

This program displays some simple things on a canvas.

```
import teek

window = teek.Window()
canvas = teek.Canvas(window)
canvas.pack()

canvas.create_line(100, 100, 120, 150)
canvas.create_oval(200, 200, 250, 250)
canvas.create_rectangle(200, 100, 230, 130)

window.on_delete_window.connect(teek.quit)
teek.run()
```

The background color of the canvas depends on the system that this code is ran on. If you don't want that, you can create the canvas like `canvas = teek.Canvas(window, bg='white')`, for instance.

The numbers passed to *`create_line()`* and similar methods are **coordinates**. They work so that, for example, `(200, 100)` means 200 pixels right from the left side of the canvas, and 100 pixels down from the top. The "how much right" coordinate is called the **x coordinate**, and the "how much down" coordinate is called the "y coordinate".

Note: The y coordinates work differently than they usually work in mathematics; that is, more y means down in Tk and teek. This also means that rotations go the other way, and positive angles mean clockwise rotations in teek even though they mean counter-clockwise rotations in mathematics.

The `create_something()` methods take four numbers each, because they are two pairs of `(x, y)` coordinates. For example, `canvas.create_rectangle(200, 100, 230, 130)` means that one corner of the created rectangle is at `(200, 100)` and another corner is at `(230, 130)`. (It is actually a square, because it is 30 pixels high and 30 pixels wide).

Item Objects

The `create_something()` methods return item objects that you can keep around and do stuff with. For example, you can do this:

```
>>> canvas = teek.Canvas(teek.Window())
>>> square = canvas.create_rectangle(200, 100, 230, 130)
>>> square
<rectangle canvas item at (200.0, 100.0, 230.0, 130.0)>
>>> square.config['fill'] = 'red'       # this makes the square red
```

Or you can pass some config options to `create_rectangle()` directly, as usual:

```
>>> square = canvas.create_rectangle(200, 100, 230, 130, fill='red')
>>> square.config['fill']
<Color 'red': red=255, green=0, blue=0>
```

Here `square` is a canvas item object.

All canvas item objects have these attributes and methods:

`some_canvas_item.`**`config`**
: This dictionary-like object is similar to the `config` attribute of widgets, which is explained *here*.

`some_canvas_item.`**`coords`**
: A tuple of coordinates of the canvas. This can be set to move an existing canvas item without having to create a new item.

`some_canvas_item.`**`tags`**
: This is a set-like object of the canvas item's *tags*, as strings. You can `.add()` and `.remove()` the tags, for example.

`some_canvas_item.`**`item_type_name`**
: The kind of the canvas item as a string, e.g. `'oval'` or `'rectangle'`.

`some_canvas_item.`**`delete`**`()`
: This deletes the canvas item. Trying to do something with the canvas item will raise an error.

  ```
  >>> circle = canvas.create_oval(200, 200, 250, 250)
  >>> circle
  <oval canvas item at (200.0, 200.0, 250.0, 250.0)>
  >>> circle.delete()
  >>> circle
  <deleted oval canvas item>
  ```

`some_canvas_item.`**`find_above`**`()`
`some_canvas_item.`**`find_below`**`()`
: These return another canvas item object. See `pathName find` and the `above` and `below` parts of `pathName addtag` in canvas(3tk) for more information.

The type of the canvas items is accessible as `the_canvas_widget.Item`. It's useful for `isinstance()` checks, but not much else:

```
>>> isinstance(circle, canvas.Item)
True
>>> isinstance('lolwat', canvas.Item)
False
```

Note that this is different for each canvas; the items of two different canvases are not of the same type.

Tags

If you have lots of related canvas items, you can just keep a list of them and do something to each item in that list. Alternatively, you can use tags to mark the canvas items, and then do something to all items tagged with a specific tag. Like this:

```
>>> canvas = teek.Canvas(teek.Window())
>>> line = canvas.create_line(100, 100, 120, 150)
>>> circle = canvas.create_oval(200, 200, 250, 250)
>>> square = canvas.create_rectangle(200, 100, 230, 130)
>>> canvas.find_withtag('lol')
[]
>>> circle.tags.add('lol')
>>> canvas.find_withtag('lol')   # returns a list that contains the circle
[<oval canvas item at (200.0, 200.0, 250.0, 250.0)>]
```

Canvas Widget Methods and Attributes

class teek.**Canvas**(**args*, ***kwargs*)
This is the canvas widget.

Manual page: canvas(3tk)

create_line(**x_and_y_coords*, ***kwargs*)
create_oval(*x1*, *y1*, *x2*, *y2*, ***kwargs*)
create_rectangle(*x1*, *y1*, *x2*, *y2*, ***kwargs*)
These create and return new *canvas items*. See the appropriate sections of canvas(3tk) for details, e.g. `RECTANGLE ITEMS` for `create_rectangle()`.

find_all()
Returns a list of all items on the canvas.

find_closest(*x*, *y*, **args*)
Returns the canvas item that is closest to the given coordinates.

See the `closest` documentation of `pathName addtag` in canvas(3tk) for details.

find_enclosed(*x1*, *y1*, *x2*, *y2*)
Returns a list of canvas items.

See the `enclosed` documentation of `pathName addtag` in canvas(3tk) for details.

find_overlapping(*x1*, *y1*, *x2*, *y2*)
Returns a list of canvas items.

See the `enclosed` documentation of `pathName addtag` in canvas(3tk) for details.

find_withtag(*tag_name*)
Returns a list of canvas items that have the given *tag*.

The tag name is given as a string. See the `enclosed` documentation of `pathName addtag` in canvas(3tk) for details.

2.2.2 Menu Widget

Manual page: menu(3tk)

You can use menu widges for a few different things:

- **Menu bars** are menus that are typically displayed at the top of a window, or top of the screen if you are using e.g. Mac OSX.
- **Pop-up menus** open when the user e.g. right-clicks something.

Here is an example of a menu bar:

```
import teek

window = teek.Window()

def hello():
    print("hello")

window.config['menu'] = teek.Menu([
    teek.MenuItem("File", [
        teek.MenuItem("New", hello),
        teek.MenuItem("Open", hello),
        teek.MenuItem("Save", hello),
        teek.MenuItem("Quit", hello),
    ]),
    teek.MenuItem("Edit", [
        teek.MenuItem("Cut", hello),
        teek.MenuItem("Copy", hello),
        teek.MenuItem("Paste", hello),
    ]),
])

window.geometry(300, 200)
window.on_delete_window.connect(teek.quit)
teek.run()
```

As you can see, *Menu* takes one argument, which is a list of *MenuItem* objects. This example uses two kinds of menu items; some menu items just call the `hello()` function when they are clicked, while the "File" and "Edit" items display submenus. There are more details about different kinds of items *below*.

Here is a pop-up menu example. See *bind documentation* for more details about the binding stuff.

```
import teek

window = teek.Window()

def hello():
    print("hello")

menu = teek.Menu([
    teek.MenuItem("Cut", hello),
    teek.MenuItem("Copy", hello),
    teek.MenuItem("Paste", hello),
])

def on_right_click(event):
    menu.popup(event.rootx, event.rooty)

if teek.windowingsystem() == 'aqua':
    # running on Mac OSX, there's no right-click so this must be done a bit
    # differently
    window.bind('<Button-2>', on_right_click, event=True)
    window.bind('<Control-Button-1>', on_right_click, event=True)
```

(continues on next page)

(continued from previous page)

```
else:
    window.bind('<Button-3>', on_right_click, event=True)

window.geometry(300, 200)
window.on_delete_window.connect(teek.quit)
teek.run()
```

I found the Mac OSX specific code from here.

Menu widgets are *not* Ttk widgets. If you don't know what that means, you should go *here* and learn. The only practical thing I can think of right now is that menus don't have a *state* attribute.

Creating Menu Items

There are a few different ways to create instances of *MenuItem*. Here `label` must be a string.

- `MenuItem()` creates a separator.
- `MenuItem(label, function)` creates a `command` item that runs `function()` when it's clicked. See also *Button*.
- `MenuItem(label, checked_var)` creates a `checkbutton` menu item. The `checked_var` must be a *BooleanVar*. See also *Checkbutton*.
- `MenuItem(label, string_var, value)` creates a `radiobutton` menu item. Use this for letting the user choose one of multiple options. Clicking the item sets `value` to `string_var`. The `string_var` must be a *StringVar* object, and `value` must be a string.
- `MenuItem(label, menu)` creates a `cascade` menu item; that is, it displays a submenu with the items of `menu` in it. The `menu` must be a *Menu* widget.
- `MenuItem(label, item_list)` is a handy way to create a new *Menu* and add it as a `cascade` item as explained above.

You can also pass options as keyword arguments in any of the above forms. The available options are documented as `MENU ENTRY OPTIONS` in menu(3tk). For example, instead of this...

```
MenuItem("Copy", do_the_copy)
```

...you probably want to do something like this:

```
MenuItem("Copy", do_the_copy, accelerator='Ctrl+C')
```

Note that this does not *bind* anything automatically, so you need to do that yourself if want that Ctrl+C actually does something.

Here is an example that demonstrates most things. See *StringVar* and *BooleanVar* documentation for more info about them.

```
import teek


def on_click():
    print("clicked")

def on_check(is_checked):
    print("is it checked now?", is_checked)
```

(continues on next page)

(continued from previous page)

```
def on_choice(choice):
    print("chose", repr(choice))


window = teek.Window()

submenu = teek.Menu([
    teek.MenuItem("Asd", on_click),
    teek.MenuItem("Toot", on_click),
])

check_var = teek.BooleanVar()
check_var.write_trace.connect(on_check)
choice_var = teek.StringVar()
choice_var.write_trace.connect(on_choice)

window.config['menu'] = teek.Menu([
    teek.MenuItem("Stuff", [
        teek.MenuItem("Click me", on_click),
        teek.MenuItem("Check me", check_var),
        teek.MenuItem("More stuff", submenu),
        teek.MenuItem(),      # separator
        teek.MenuItem("Choice 1", choice_var, "one"),
        teek.MenuItem("Choice 2", choice_var, "two"),
        teek.MenuItem("Choice 3", choice_var, "three"),
    ]),
])

window.geometry(300, 200)
window.on_delete_window.connect(teek.quit)
teek.run()
```

Reference

class teek.**Menu**(*items=()*, ***kwargs*)

This is the menu widget.

The `items` should be an iterable of *MenuItem* objects, and it's treated so that this...

```
menu = teek.Menu([
    teek.MenuItem("Click me", print),
    teek.MenuItem("No, click me instead", print),
])
```

...does the same thing as this:

```
menu = teek.Menu()
menu.append(teek.MenuItem("Click me", print))
menu.append(teek.MenuItem("No, click me instead", print))
```

Menu widgets behave like lists of menu items, so if you can do something to a list of *MenuItem* objects, you can probably do it directly to a *Menu* widget as well.

However, menu widgets don't support slicing, like lists do:

```
>>> menu = teek.Menu([
...     teek.MenuItem("Click me", print),
... ])
>>> menu.append(teek.MenuItem("No, click me instead", print))
>>> menu
<teek.Menu widget: contains 2 items>
>>> menu[0]     # this works
<MenuItem('Click me', <built-in function print>): type='command', added to a menu>
>>> for item in menu:   # this works
...     print(item)
...
<MenuItem('Click me', <built-in function print>): type='command', added to a menu>
<MenuItem('No, click me instead', <built-in function print>): type='command',␣
↪added to a menu>
>>> menu[:2]    # but this doesn't work
Traceback (most recent call last):
  ...
TypeError: slicing a Menu widget is not supported
>>> list(menu)[:2]    # workaround      # doctest: +ELLIPSIS
[<MenuItem(...): ...>, <MenuItem(...): ...>]
```

Menu objects assume that nothing changes the underlying Tk menu widget without the *Menu* object. For example:

```
>>> menu = teek.Menu()
>>> command = menu.to_tcl()
>>> command      # doctest: +SKIP
'.menu1'
>>> # DON'T DO THIS, this is a bad idea
>>> teek.tcl_eval(None, '%s add checkbutton -command {puts hello}' % command)
>>> len(menu)    # the menu widget doesn't know that we added an item
0
```

If you don't know what *tcl_eval()* does, you don't need to worry about doing this accidentally.

Manual page: menu(3tk)

popup(*x*, *y*, *menu_item=None*)

Displays the menu on the screen.

x and y are coordinates in pixels, relative to the screen. See tk_popup(3tk) for details. If *menu_item* is given, its index is passed to tk_popup(3tk).

There are two ways to show popup menus in Tk. This is one of them, and `post` is another. I spent a while trying to find something that explains the difference, and the best thing I found is this book. The book uses `tk_popup`, and one of the authors is John Ousterhout, the creator of Tcl and Tk.

class teek.**MenuItem**(**args*, ***kwargs*)

Represents an item of a menu. See *Creating Menu Items* for details about the arguments.

Tk's manual pages call these things "menu entries" instead of "menu items", but I called them items to avoid confusing these with *Entry*.

There are two kinds of *MenuItem* objects:

- Menu items that are not in any *Menu* widget because they haven't been added to a menu yet, or they have been removed from a menu. Trying to do something with these menu items will likely raise a `RuntimeError`.
- Menu items that are currently in a *Menu*.

Here's an example:

```
>>> item = teek.MenuItem("Click me", print)
>>> item.config['label'] = "New text"
Traceback (most recent call last):
    ...
RuntimeError: the MenuItem hasn't been added to a Menu yet
>>> menu = teek.Menu()
>>> menu.append(item)
>>> item.config['label'] = "New text"
>>> item.config['label']
'New text'
```

config
: This attribute is similar to *Widget.config*. See `MENU ENTRY OPTIONS` in menu(3tk).

  The types of the values are the same as for similar widgets. For example, the `'command'` of a *Button* widget is a *Callback* object connected to a function passed to *Button*, and so is the `'command'` of `teek.MenuItem("Click me", some_function)`.

type
: This is a string. Currently the possible values are `'separator'`, `'checkbutton'`, `'command'`, `'cascade'` and `'radiobutton'` as documented *above*. Don't set this attribute yourself.

2.2.3 Notebook Widget

Manual page: ttk_notebook(3tk)

This widget is useful for creating tabs as in the tabs of your web browser, not `\t` characters. Let's look at an example.

```
import teek

window = teek.Window("Notebook Example")
notebook = teek.Notebook(window)
notebook.pack(fill='both', expand=True)

for number in [1, 2, 3]:
    label = teek.Label(notebook, "Hello {}!".format(number))
    tab = teek.NotebookTab(label, text="Tab {}".format(number))
    notebook.append(tab)

window.geometry(300, 200)
window.on_delete_window.connect(teek.quit)
teek.run()
```

This program displays a notebook widget with 3 tabs. Let's go trough some of the code.

```
label = teek.Label(notebook, "Hello {}!".format(number))
```

The label is created as a child widget of the notebook, because it will be added to the notebook eventually. However, we *don't* use a *geometry manager* because the notebook itself handles showing the widget.

```
tab = teek.NotebookTab(label, text="Tab {}".format(number))
```

We need to create a *NotebookTab* object in order to add a tab to the notebook. The *NotebookTab* objects themselves are **not** widgets. A *NotebookTab* represents a widget and its tab options like `text`.

```
notebook.append(tab)
```

Lists also have an append method, and this is no coincidence. *Notebook* widgets behave like lists, and if you can do something to a list, you can probably do it to a notebook as well. However, there are a few things you can't do to notebook widgets, but you can do to lists:

- You can't put any objects you want into a *Notebook*. All the objects must be *NotebookTabs*.
- You can't slice notebooks like `notebook[1:]`. However, you can get a list of all tabs in the notebook with `list(notebook)` and then slice that.
- You can't sort notebooks like `notebook.sort()`. However, you can do `sorted(notebook)` to get a sorted list of *NotebookTabs*, except that it doesn't quite work because tab objects can't be compared with each other. Something like `sorted(notebook, key=(lambda tab: tab.config['text']))` might be useful though.
- You can't add the same *NotebookTab* to the notebook twice, and in fact, you can't create two *NotebookTabs* that represent the same widget, so you can't add the same widget to the notebook as two different tabs.

Note that instead of this...

```
label = teek.Label(notebook, "Hello {}!".format(number))
tab = teek.NotebookTab(label, text="Tab {}".format(number))
notebook.append(tab)
```

...you can also do this:

```
notebook.append(
    teek.NotebookTab(
        teek.Label(notebook, "Hello {}!".format(number)),
        text="Tab {}".format(number)
    )
)
```

I recommend using common sense here. The first alternative is actually only half as many lines of code as the second one, even though it uses more variables. Try to keep the code readable, as usual.

Here is some reference:

class teek.**Notebook**(*parent*, ***kwargs*)

This is the notebook widget.

If you try to add a tab that is already in the notebook, that tab will be moved. For example:

```
>>> notebook = teek.Notebook(teek.Window())
>>> tab1 = teek.NotebookTab(teek.Label(notebook, text="1"), text="One")
>>> tab2 = teek.NotebookTab(teek.Label(notebook, text="2"), text="Two")
>>> notebook.extend([tab1, tab2])
>>> list(notebook)       # doctest: +NORMALIZE_WHITESPACE
[NotebookTab(<teek.Label widget: text='1'>, text='One'),
 NotebookTab(<teek.Label widget: text='2'>, text='Two')]
>>> notebook.append(notebook[0])
>>> list(notebook)       # doctest: +NORMALIZE_WHITESPACE
[NotebookTab(<teek.Label widget: text='2'>, text='Two'),
 NotebookTab(<teek.Label widget: text='1'>, text='One')]
```

For doing advanced magic, you can create a new class that inherits from *Notebook*. Here are some facts that can be useful when deciding which methods to override:

- Override `__delitem__()` to customize removing tabs from the notebook. A deletion like `del notebook[index]` does `notebook.__delitem__(index)`, which calls `pathName forget` documented in ttk_notebook(3tk). All other kinds of deletions call `__delitem__` as well.
- Override `insert()` if you want to customize adding new tabs to the notebook. The `insert` method is called every time when a new tab is added with any method. Make sure that your override is compatible with the `insert()` method of `collections.abc.MutableSequence`, and make sure that only the order of the tabs changes if the new tab is already in the notebook.
- Bind to `<<NotebookTabChanged>>` if you want to customize what happens when a different tab is selected. That runs when the user changes a tab or the tab is changed with the *`selected_tab`* property. `<<NotebookTabChanged>>` is documented in the `VIRTUAL EVENTS` section of ttk_notebook(3tk).

As usual, use `super()` when overriding.

Manual page: ttk_notebook(3tk)

`append_and_select`(*tab*)
A convenient way to add a tab to the notebook and select it.

`notebook.append_and_select(tab)` is same as:

```
notebook.append(tab)
notebook.selected_tab = tab
```

`get_tab_by_widget`(*widget*)
Finds a *`NotebookTab`* object by the *`widget`* attribute.

If there is no tab with the given widget, a new tab is created.

```
>>> notebook = teek.Notebook(teek.Window())
>>> label = teek.Label(notebook, text='lol')
>>> tab = teek.NotebookTab(label)
>>> notebook.append(tab)
>>> tab
NotebookTab(<teek.Label widget: text='lol'>)
>>> notebook.get_tab_by_widget(label)
NotebookTab(<teek.Label widget: text='lol'>)
```

`move`(*tab*, *new_index*)
Move a tab so that after calling this, `self[new_index]` is `tab`.

The new index may be negative. `IndexError` is raised if the index is not in the correct range.

`selected_tab`
This is the tab that the user is currently looking at.

This is `None` if there are no tabs in the notebook. You can set this to any other tab in the notebook to change the currently selected tab.

`class` `teek.`**`NotebookTab`**(*widget*, ***kwargs*)
Represents a tab that is in a notebook, or is ready to be added to a notebook.

The *widget* must be a child widget of a *`Notebook`* widget. Each *`NotebookTab`* belongs to the widget's parent notebook; for example, if you create a tab like this...

```
tab = teek.NotebookTab(teek.Label(asd_notebook, "hello"))
```

...then the tab cannot be added to any other notebook widget than `asd_notebook`, because `asd_notebook` is the parent widget of the label.

Most methods raise `RuntimeError` if the tab has not been added to the notebook yet. This includes doing pretty much anything with *`config`*.

For convenience, options can be passed when creating a *`NotebookTab`*, so that this...

```
notebook.append(teek.NotebookTab(some_widget, text="Tab Title"))
```

...does the same thing as this:

```
tab = teek.NotebookTab(some_widget, text="Tab Title")
notebook.append(tab)
tab.config['text'] = "Tab Title"
```

There are never multiple *`NotebookTab`* objects that represent the same tab.

config
: Similar to the `config` attribute that widgets have. The available options are documented as `TAB OPTIONS` in ttk_notebook(3tk). Attempting to use this raises `RuntimeError` if the tab hasn't been added to the notebook yet.

widget
: This attribute and initialization argument is the widget in the tab. It should be a child widget of the notebook. Use `tab.widget.parent` to access the *`Notebook`* that the tab belongs to.

initial_options
: A dict of keyword arguments passed to NotebookTab. When the tab is added to the notebook for the first time, *`config`* is updated from this dict.

hide()
: Call `pathName hide` documented in ttk_notebook(3tk).

  Use *`unhide()`* to make the tab visible again. `RuntimeError` is raised if the tab has not been added to a notebook.

unhide()
: Undo a *`hide()`* call.

2.2.4 Text Widget

Manual page: text(3tk)

This widget is useful for displaying text that the user should be able to edit. The text can contain multiple lines. Use *`Entry`* if the text is just 1 line, or *`Label`* if you don't want the user to be able to edit the text.

Note: This widget is **not** a Ttk widget because there is no Ttk text widget. However, that's not a problem because usually text widgets are configured to have custom colors anyway instead of using the Ttk theme's colors. For example, my editor lets the user change the colors of the text widget by selecting a color theme from a menu. Only the text widget colors change, and Ttk widgets are not affected by that.

Hello World!

```
import teek

window = teek.Window()
text = teek.Text(window)
```

(continues on next page)

(continued from previous page)

```
text.pack()
text.insert(text.start, 'Hello World!')

window.on_delete_window.connect(teek.quit)
teek.run()
```

This program displays a window with a `Hello World!` text in it displayed using a monospace font. It lets you edit the text.

There is more example code in examples/text.py.

TextIndex Objects

In the above example, we used *`Text.insert()`*, and we told it to insert the hello world to the beginning of the text widget by passing it `text.start`. Here `text.start` was a text index `namedtuple`:

```
>>> window = teek.Window()
>>> text = teek.Text(window)
>>> text.start
TextIndex(line=1, column=0)
```

Note: Line numbers start at 1, and columns start at 0. Tk does this too, and teek doesn't hide it because it is very common to call the first line "line 1"...

```
Traceback (most recent call last):
  File "my_file.py", line 1, in <module>
    first line of code
  ...
```

...but it's just as common to start column numbering at 0, just like string indexing in general.

Avoid working with 0-based line numbers if you can. That way your teek code will probably be simpler and less confusing.

The text indices can do everything that namedtuples can do.

```
>>> text.start.line     # do this if you need only line or column
1
>>> text.start.column
0
>>> line, column = text.start       # do this if you need both
>>> line
1
>>> column
0
```

In general, they behave like tuples:

```
>>> for lol in text.start:
...     print(lol)
...
1
0
```

(continues on next page)

(continued from previous page)

```
>>> list(text.start)
[1, 0]
```

There is also `text.end`, which is the text index of the end of the text widget:

```
>>> text.end
TextIndex(line=1, column=0)
>>> text.end == text.start      # there's nothing in this text widget yet
True
>>> text.insert(text.start, 'hello')
>>> text.end
TextIndex(line=1, column=5)
>>> text.end == text.start
False
>>> text.insert(text.end, '\nsome text')    # now there are 2 lines of text
>>> text.end
TextIndex(line=2, column=9)
>>> text.get(text.start, text.end)
'hello\nsome text'
```

The indexes may be out of bounds, but that does not create errors:

```
>>> text.TextIndex(1000, 1000)
TextIndex(line=1000, column=1000)
>>> text.get(text.start, text.TextIndex(1000, 1000))
'hello\nsome text'
>>> text.TextIndex(1000, 1000).between_start_end()
TextIndex(line=2, column=9)
```

The text index class can be accessed as `some_text_widget.TextIndex`:

```
>>> text.TextIndex(1, 0)
TextIndex(line=1, column=0)
>>> isinstance('lol', text.TextIndex)
False
```

The `TextIndex` classes are also valid *type specifications*.

Text indices have the following attributes and methods:

`some_text_index.`**`forward`**(*chars=0*, *indices=0*, *lines=0*)

Return a new text index after this index (or before, if the arguments are negative, or the same index if all arguments are zero). All arguments must be integers, and given as keyword arguments. Search for `+ count` in text(3tk) for an explanation of what each argument does.

```
>>> text.get(text.start, text.end)
'hello\nsome text'
>>> text.start.forward(chars=3)
TextIndex(line=1, column=3)
>>> text.start.forward(chars=7)       # goes over a newline character
TextIndex(line=2, column=1)
```

The returned indexes are always converted to be between *`start`* and *`end`* with *`between_start_end()`*:

```
>>> text.start.forward(chars=1000)     # won't go past end of text widget
TextIndex(line=2, column=9)
```

some_text_index.**back**(*chars=0*, *indices=0*, *lines=0*)
Like *forward()*, but goes back instead of forward.

some_text_index.**linestart**()
some_text_index.**lineend**()
some_text_index.**wordstart**()
some_text_index.**wordend**()
These return new text indices. Search for e.g. `linestart` in text(3tk) for details.

some_text_index.**between_start_end**()
If the text index is before *start* or after *end*, this returns *start* or *end*, respectively. Otherwise the text index is returned as is.

```
>>> text.TextIndex(1000, 1000)
TextIndex(line=1000, column=1000)
>>> text.TextIndex(1000, 1000).between_start_end()
TextIndex(line=2, column=9)
>>> text.end
TextIndex(line=2, column=9)
```

Tip: Text indices are usually namedtuples, but methods that take text indices as arguments (e.g. *insert()*) can also take regular `(line, column)` tuples.

Marks

If you have a text widget that contains `hello world`, the position just before `w` changes if you change `hello` to something else, or if the user edits the text widget's content:

```
>>> text.replace(text.start, text.end, "hello world")
>>> text.get(text.start, text.end)
'hello world'
>>> before_w = text.TextIndex(1, 6)
>>> before_w
TextIndex(line=1, column=6)
>>> text.get(before_w, text.end)
'world'
>>> text.replace(text.start, text.start.forward(chars=5), 'hi')
>>> text.get(text.start, text.end)      # hello was replaced with hi
'hi world'
>>> text.get(before_w, text.end)        # but before_w didn't update!
'ld'
>>> before_w
TextIndex(line=1, column=6)
```

We can solve this problem by adding a **mark**:

```
>>> text.replace(text.start, text.end, "hello world")
>>> text.marks['before_w'] = text.TextIndex(1, 6)
>>> text.get(text.marks['before_w'], text.end)
'world'
>>> text.replace(text.start, text.start.forward(chars=5), 'hi')
>>> text.get(text.marks['before_w'], text.end)
'world'
>>> text.get(text.start, text.end)
'hi world'
```

Marks move with the text as the text before them is changed. *Text.marks* is a dictionary-like object with mark name strings as keys and *index objects* as values. There is also a special `'insert'` mark that represents the cursor position:

```
# move cursor to new_cursor_pos
text.marks['insert'] = new_cursor_pos
```

There are more details about marks in the `MARKS` section of text(3tk).

Tags

You can use tags to change things like color of *some* of the text without changing all of it. For example, this code displays a `Hello World` with a red `Hello` and a green `World`:

```
import teek

window = teek.Window("Text Widget Demo")

text = teek.Text(window)
text.pack(fill='both', expand=True)
text.insert(text.start, "hello world")

hello_tag = text.get_tag('hello_tag')
hello_tag['foreground'] = teek.Color('red')
hello_tag.add(text.start, text.start.forward(chars=len('hello')))

world_tag = text.get_tag('world_tag')
world_tag['foreground'] = teek.Color('green')
world_tag.add(text.end.back(chars=len('world')), text.end)

window.on_delete_window.connect(teek.quit)
teek.run()
```

Each tag has a name which is mostly useful for debugging. If you want to create a tag, call *get_tag()* with whatever name you want; a new tag is created if a tag with the given name doesn't exist yet.

```
>>> text.get_tag('hello_tag')
<Text widget tag 'hello_tag'>
```

Tag objects behave like *config* objects, so you can e.g. change their options with their dictionary-like behaviour. See `TAGS` in text(3tk) for a list of supported options. As usual, the options are given without a leading `-`, like `'foreground'` instead of `'-foreground'`.

There is a special tag named `'sel'` that represents the currently selected text.

Tag objects have these attributes and methods. Search for `pathName tag` in text(3tk) for more information about them.

`some_tag.`**`name`**
: The Tk name of the tag, as a string.

`some_tag.`**`add`**`(`*index1*`, `*index2*`)`
: Add this tag to text between the given *indices*.

`some_tag.`**`delete`**`()`
: Remove this tag from everywhere in the text widget, and forget all configuration options the tag has been given.

Note: *delete()* and *remove()* do different things.

---

some_tag.**remove**(*index1=None*, *index2=None*)
Remove this tag from the text widget between the given *indices*. index1 defaults to *start*, and index2 defaults to *end*.

some_tag.**to_tcl**()
Returns *some_tag.name*. See *Python to Tcl conversion*.

some_tag.**ranges**()
Returns a list of (start_index, end_index) pairs that describe where the tag has been added. The indexes are *index objects*.

some_tag.**prevrange**(*index1*, *index2=None*)
some_tag.**nextrange**(*index1*, *index2=None*)
These return the previous or next (start_index, end_index) pair. See *ranges()* and text(3tk).

some_tag.**bind**(*sequence*, *func*, *, *event=False*)

some_tag.**bindings**
These allow you to do tag-specific *bindings*.

some_tag.**lower**(*other_tag=None*)
some_tag.**raise_**(*other_tag=None*)
These call tag lower and tag raise documented in text(3tk). The raise method is called raise_ instead of raise because raise is not a valid method name in Python.

For example, if you have code like this...

```
import teek

text = teek.Text(teek.Window())
text.pack()

red_tag = text.get_tag('red')
blue_tag = text.get_tag('blue')
red_tag['foreground'] = 'red'
blue_tag['foreground'] = 'blue'

text.insert(text.start, 'asdasdasd', [red_tag, blue_tag])
```

...then the asdasdasd text has two tags that specify different foreground colors. If you want red asdasdasd, the red tag should be above the blue tag, so you need to do red_tag.raise_(blue_tag) or blue_tag.lower(red_tag).

Text Widget Methods and Attributes

class teek.**Text**(*parent*, ***kwargs*)
This is the text widget.

Manual page: text(3tk)

start
end
TextIndex objects that represents the start and end of the text.

Tip: Use `textwidget.end.line` to count the number of lines of text in the text widget.

Note that `end` changes when the text widget's content changes:

```
>>> window = teek.Window()
>>> text = teek.Text(window)
>>> text.end
TextIndex(line=1, column=0)
>>> old_end = text.end
>>> text.insert(text.end, 'hello')
>>> text.end
TextIndex(line=1, column=5)
>>> old_end
TextIndex(line=1, column=0)
>>> text.get(old_end, text.end)
'hello'
```

Tk has a concept of an invisible newline character at the end of the widget. In pure Tcl or in tkinter, getting the text from beginning to `end` returns the text in the widget plus a `\n`, which is why you almost always need to do `end - 1 char` instead of just `end`. **Teek doesn't do that** because 99% of the time it's not useful and 1% of the time it's confusing to people reading the code, so `text.get(text.start, text.end)` doesn't return anything that is not visible in the text widget.

marks
A dictionary-like object with mark names as keys and *index objects* as values. See *Marks*.

xview(**args*)
yview(**args*)
These call `pathName xview` and `pathName yview` as documented in text(3tk). Pass string arguments to these methods to invoke the subcommands. For example, `text_widget.yview('moveto', 1)` scrolls to end vertically.

If no arguments are given, these methods return a two-tuple of floats (see the manual page); otherwise, None is returned.

delete(*index1*, *index2*)
See text(3tk) and *insert()*.

get(*index1=None*, *index2=None*)
Return text in the widget.

If the indexes are not given, they default to the beginning and end of the text widget, respectively.

get_all_tags(*index=None*)
Return all tags as tag objects.

See `pathName tag names` in text(3tk) for more details.

get_tag(*name*)
Return a tag object by name, creating a new one if needed.

insert(*index*, *text*, *tag_list=()*)
Add text to the widget.

The `tag_list` can be any iterable of tag name strings or tag objects.

replace(*index1*, *index2*, *new_text*, *tag_list=()*)
See text(3tk) and *insert()*.

see (*index*)
Scroll so that an index is visible.

See text(3tk) for details.

class teek.**Widget** (*parent*, ***kwargs*)
This is a base class for all widgets.

All widgets inherit from this class, and they have all the attributes and methods documented here.

Don't create instances of `Widget` yourself like `Widget(...)`; use one of the classes documented below instead. However, you can use `Widget` with `isinstance()`; e.g. `isinstance(thingy, teek.Widget)` returns `True` if `thingy` is a teek widget.

config
A dict-like object that represents the widget's options.

```
>>> window = teek.Window()
>>> label = teek.Label(window, text='Hello World')
>>> label.config
<a config object, behaves like a dict>
>>> label.config['text']
'Hello World'
>>> label.config['text'] = 'New Text'
>>> label.config['text']
'New Text'
>>> label.config.update({'text': 'Even newer text'})
>>> label.config['text']
'Even newer text'
>>> import pprint
>>> pprint.pprint(dict(label.config))  # prints everything nicely          #␣
↪doctest: +ELLIPSIS
{...,
 'text': 'Even newer text',
 ...}
```

state
Represents the Ttk state of the widget. The state object behaves like a `set` of strings. For example, `widget.state.add('disabled')` makes a widget look like it's grayed out, and `widget.state.remove('disabled')` undoes that. See `STATES` in ttk_intro(3tk) for more details about states.

Note: Only Ttk widgets have states, and this attribute is set to None for non-Ttk widgets. If you don't know what Ttk is, you should read about it in *the teek tutorial*. Most teek widgets are ttk widgets, but some aren't, and that's mentioned in the documentation of those widgets.

---

tk_class_name
Tk's class name of the widget class, as a string.

This is a class attribute, but it can be accessed from instances as well:

```
>>> text = teek.Text(teek.Window())
>>> text.tk_class_name
'Text'
>>> teek.Text.tk_class_name
'Text'
```

Note that Tk's class names are sometimes different from the names of Python classes, and this attribute can also be None in some special cases.

```
>>> teek.Label.tk_class_name
'TLabel'
>>> class AsdLabel(teek.Label):
...     pass
...
>>> AsdLabel.tk_class_name
'TLabel'
>>> print(teek.Window.tk_class_name)
None
>>> print(teek.Widget.tk_class_name)
None
```

command_list
A list of command strings from *create_command()*.

Append a command to this if you want the command to be deleted with *delete_command()* when the widget is destroyed (with e.g. *destroy()*).

destroy()
Delete this widget and all child widgets.

Manual page: destroy(3tk)

Note: Don't override this in a subclass. In some cases, the widget is destroyed without a call to this method.

```
>>> class BrokenFunnyLabel(teek.Label):
...     def destroy(self):
...         print("destroying")
...         super().destroy()
...
>>> BrokenFunnyLabel(teek.Window()).pack()
>>> teek.quit()
>>> # nothing was printed!
```

Use the `<Destroy>` event instead:

```
>>> class WorkingFunnyLabel(teek.Label):
...     def __init__(self, *args, **kwargs):
...         super().__init__(*args, **kwargs)
...         self.bind('<Destroy>', self._destroy_callback)
...     def _destroy_callback(self):
...         print("destroying")
...
>>> WorkingFunnyLabel(teek.Window()).pack()
>>> teek.quit()
destroying
```

---

focus(*, *force=False*)
Focuses the widget with focus(3tk).

If `force=True` is given, the `-force` option is used.

classmethod from_tcl(*path_string*)
Creates a widget from a Tcl path name.

In Tcl, widgets are represented as commands, and doing something to the widget invokes the command. Use this method if you know the Tcl command and you would like to have a widget object instead.

This method raises `TypeError` if it's called from a different `Widget` subclass than what the type of the `path_string` widget is:

```
>>> window = teek.Window()
>>> teek.Button.from_tcl(teek.Label(window).to_tcl())  # doctest: +ELLIPSIS
Traceback (most recent call last):
    ...
TypeError: '...' is a Label, not a Button
```

to_tcl()
Returns the widget's Tcl command name. See *from_tcl()*.

winfo_children()
Returns a list of child widgets that this widget has.

Manual page: winfo(3tk)

winfo_exists()
Returns False if the widget has been destroyed. See *destroy()*.

Manual page: winfo(3tk)

winfo_height()
Calls `winfo height`. Returns an integer.

Manual page: winfo(3tk)

winfo_id()
Calls `winfo id`. Returns an integer.

Manual page: winfo(3tk)

winfo_reqheight()
Calls `winfo reqheight`. Returns an integer.

Manual page: winfo(3tk)

winfo_reqwidth()
Calls `winfo reqwidth`. Returns an integer.

Manual page: winfo(3tk)

winfo_rootx()
Calls `winfo rootx`. Returns an integer.

Manual page: winfo(3tk)

winfo_rooty()
Calls `winfo rooty`. Returns an integer.

Manual page: winfo(3tk)

winfo_toplevel()
Returns the *Toplevel* widget that this widget is in.

Manual page: winfo(3tk)

winfo_width()
Calls `winfo width`. Returns an integer.

Manual page: winfo(3tk)

winfo_x()
Calls `winfo x`. Returns an integer.

Manual page: winfo(3tk)

winfo_y()
Calls `winfo y`. Returns an integer.

Manual page: winfo(3tk)

class `teek.`**Button**(*parent*, *text=''*, *command=None*, ***kwargs*)
A widget that runs a callback when it's clicked.

See examples/button.py for example code.

`text` can be given as with *`Label`*. The `'command'` option is not settable, and its value is a *`Callback`* that runs with no arguments when the button is clicked. If the *command* argument is given, it will be treated so that this...

```
button = teek.Button(some_widget, "Click me", do_something)
```

...does the same thing as this:

```
button = teek.Button(some_widget, "Click me")
button.config['command'].connect(do_something)
```

See *`Callback.connect()`* documentation if you need to pass arguments to the `do_something` function.

Manual page: ttk_button(3tk)

invoke()
Runs the command callback.

See `pathname invoke` in ttk_button(3tk) for details.

class `teek.`**Canvas**
See *Canvas Widget*.

class `teek.`**Checkbutton**(*parent*, *text=''*, *command=None*, ***kwargs*)
A square-shaped, checkable box with text next to it.

See examples/checkbutton.py for example code.

For convenience, `text` and `command` arguments work the same way as with *`Button`*.

The `'command'` option is not settable, and its value is a *`Callback`*. By default, it runs with `True` as the only argument when the checkbutton is checked, and with `False` when the checkbutton is unchecked. You can pass `onvalue=False, offvalue=True` to reverse this if you find it useful for some reason. This also affects the values that end up in the `'variable'` option (see manual page), which is a *`BooleanVar`*.

Manual page: ttk_checkbutton(3tk)

invoke()
Checks or unchecks the checkbutton, updates the variable and runs the command callback.

See `pathname invoke` in ttk_checkbutton(3tk) for details.

class `teek.`**Combobox**(*parent*, *text=''*, ***kwargs*)
An entry that displays a list of valid values.

This class inherits from *`Entry`*, so it has all the attributes and methods of *`Entry`*, like *`text`* and *`cursor_pos`*.

Manual page: ttk_combobox(3tk)

class teek.**Entry**(*parent*, *text=''*, ***kwargs*)
A widget for asking the user to enter a one-line string.

The `text` option works as with *Label*.

See also:

Use *Label* if you want to display text without letting the user edit it. Entries are also not suitable for text with more than one line; use *Text* instead if you want multiple lines.

Manual page: ttk_entry(3tk)

cursor_pos
The integer index of the cursor in the entry, so that `entry.text[:entry.cursor_pos]` and `entry.text[entry.cursor_pos:]` are the text before and after the cursor, respectively.

You can set this attribute to move the cursor.

text
The string of text in the entry widget.

Setting and getting this attribute calls `get`, `insert` and `delete` documented in ttk_entry(3tk).

class teek.**Frame**(*parent*, ***kwargs*)
An empty widget. Frames are often used as containers for other widgets.

Manual page: ttk_frame(3tk)

class teek.**Label**(*parent*, *text=''*, ***kwargs*)
A widget that displays text.

For convenience, the `text` option can be also given as a positional initialization argument, so `teek.Label(parent, "hello")` and `teek.Label(parent, text="hello")` do the same thing.

Manual page: ttk_label(3tk)

class teek.**LabelFrame**(*parent*, *text=''*, ***kwargs*)
A frame with a visible border line and title text.

For convenience, the `text` option can be given as with *Label*.

Manual page: ttk_labelframe(3tk)

class teek.**Menu**
See *Menu Widget*.

class teek.**Notebook**
See *Notebook Widget*.

class teek.**Progressbar**(*parent*, ***kwargs*)
Displays progress of a long-running operation. This is useful if you are *running something concurrently* and you want to let the user know that something is happening.

The progress bar can be used in two modes. Pass `mode='indeterminate'` and call *start()* to make the progress bar bounce back and forth forever. If you want to create a progress bar that actually displays progress instead of just letting the user know that something is happening, don't pass `mode='indeterminate'`; the default is `mode='determinate'`, which does what you want.

There's a `'value'` option that can be used to set the progress in determinate mode. A value of 0 means that nothing is done, and 100 means that we are ready. If you do math on a regular basis, that's all you need to know, but if you are not very good at math, keep reading:

Progress Math

If your program does 5 things, and 2 of them are done, you should do this:

```
progress_bar.config['value'] = (2 / 5) * 100
```

It works like this:

- The program has done 2 things out of 5; that is, 2/5. That is a division. Its value turns out to be 0.4.
- We want percents. They are numbers between 0 and 100. The `done / total` calculation gives us a number between 0 and 1; if we have done nothing, we have `0 / 5 == 0.0`, and if we have done everything, we have `5 / 5 == 1.0`. If we add `* 100`, we get `0.0 * 100 = 0.0` when we haven't done anything, and `1.0 * 100 == 100.0` when we have done everything. Awesome!

```
progress_bar.config['value'] = (done / total) * 100
```

However, this fails if `total == 0`:

```
>>> 1/0
Traceback (most recent call last):
    ...
ZeroDivisionError: division by zero
```

If we have no work to do and we have done nothing (`0/0`), then how many percents of the work is done? It doesn't make sense. You can handle these cases e.g. like this:

```
if total == 0:
    # 0/0 things done, make the progress bar grayed out because
    # there is no progress to indicate
    progress_bar.state.add('disabled')
else:
    progress_bar.config['value'] = (done / total) * 100
```

If multiplying by 100 is annoying, you can create the progress bar like this...

```
progress_bar = teek.Progressbar(parent_widget, maximum=1)
```

...and then set numbers between 0 and 1 to `progress_bar.config['value']`:

```
if total == 0:
    progress_bar.state.add('disabled')
else:
    progress_bar.config['value'] = done / total
```

Manual page: ttk_progressbar(3tk)

start(*interval=50*)
: Makes an indeterminate mode progress bar bounce back and forth.

  The progress bar will move by a tiny bit every *interval* milliseconds. A small interval makes the progress bar look smoother, but don't make it too small to avoid keeping CPU usage down. The default should be good enough for most things.

stop()
: Stops the bouncing started by *start()*.

class teek.**Scrollbar**(*parent*, ***kwargs*)
: A widget for scrolling other widgets, like *Text*.

  In order to use a scrollbar, there are two things you need to do:

1. Tell a scrollable widget (e.g. *Text*) to use the scrollbar.
2. Tell the scrollbar to scroll the widget.

For example:

```
import teek

window = teek.Window()

text = teek.Text(window)
text.pack(side='left', fill='both', expand=True)
scrollbar = teek.Scrollbar(window)
scrollbar.pack(side='left', fill='y')

text.config['yscrollcommand'].connect(scrollbar.set)   # 1.
scrollbar.config['command'].connect(text.yview)        # 2.

window.on_delete_window.connect(teek.quit)
teek.run()
```

The value of the scrollbar's `'command'` option is a *Callback* that runs when the scrollbar is scrolled. It runs with arguments suitable for *Text.xview()* or *Text.yview()*. See `SCROLLING COMMANDS` in ttk_scrollbar(3tk) for details about the arguments.

Manual page: ttk_scrollbar(3tk)

get()
: Return a two-tuple of floats that have been passed to *set()*.

  See also `pathName get` in ttk_scrollbar(3tk).

set(*first*, *last*)
: Set the scrollbar's position.

  See `pathName set` in ttk_scrollbar(3tk) for details.

class `teek.`**Separator**(*parent*, ***kwargs*)
: A horizontal or vertical line, depending on an `orient` option.

  Create a horizontal separator like this...

```
separator = teek.Separator(some_widget, orient='horizontal')
separator.pack(fill='x')    # default is side='top'
```

  ...and create a vertical separator like this:

```
separator = teek.Separator(some_widget, orient='vertical')
separator.pack(fill='y', side='left')   # can also use side='right'
```

  See examples/separator.py for more example code.

  Manual page: ttk_separator(3tk)

class `teek.`**Spinbox**(*parent*, *, *command=None*, ***kwargs*)
: An entry with up and down buttons.

  This class inherits from *Entry*, so it has all the attributes and methods of *Entry*, like *text* and *cursor_pos*.

  The value of the `'command'` option is a *Callback* that is ran with no arguments. If a `command` keyword argument is given, it will be connected to the callback automatically.

Manual page: ttk_spinbox(3tk)

class teek.**Text**
See *Text Widget*.

class teek.**Toplevel**(*title=None*, ***options*)
This represents a *non-Ttk* `toplevel` widget.

Usually it's easiest to use *Window* instead. It behaves like a `Toplevel` widget, but it's actually a `Toplevel` with a `Frame` inside it.

Manual page: toplevel(3tk)

geometry(*width=None*, *height=None*, *x=None*, *y=None*)
Set or get the size and place of the window in pixels.

Tk's geometries are strings like `'100x200+300+400'`, but that's not very pythonic, so this method works with integers and namedtuples instead. This method can be called in a few different ways:

- If *width* and *height* are given, the window is resized.
- If *x* and *y* are given, the window is moved.
- If all arguments are given, the window is resized and moved.
- If no arguments are given, the current geometry is returned as a namedtuple with *width*, *height*, *x* and *y* fields.
- Calling this method otherwise raises an error.

Examples:

```
>>> import teek
>>> window = teek.Window()
>>> window.geometry(300, 200)     # resize to 300px wide, 200px high
>>> window.geometry(x=0, y=0)     # move to upper left corner
>>> window.geometry()
Geometry(width=300, height=200, x=0, y=0)
>>> window.geometry().width
300
```

See also `wm geometry` in wm(3tk).

iconphoto(**images*, *default=False*)
Calls `wm iconphoto` documented in wm(3tk).

Positional arguments should be *Image* objects. If `default=True` is given, the `-default` switch is used; otherwise it isn't.

title
wm_state
transient
minsize
maxsize

withdraw()
iconify()
deiconify()
These attributes and methods correspond to similarly named things in wm(3tk). Note that `wm_state` is `state` in the manual page; the teek attribute is `wm_state` to make it explicit that it is the wm state, not some other state.

All of the attributes are settable, so you can do e.g. `my_toplevel.title = "lol"`. Here are the types of the attributes:

- `title` and `wm_state` are strings.
- `transient` is a widget.
- `minsize` and `maxsize` are tuples of two integers.

Note: If `transient` is set to a *`Window`*, looking it up won't give back that same window; instead, it gives the *`toplevel`* of the window.

---

wait_window()
Waits until the window is destroyed with *`destroy()`*.

This method blocks until the window is destroyed, but it can still be called from the *event loop*; behind the covers, it runs another event loop that makes the GUI not freeze.

See `tkwait window` in tkwait(3tk) for more details.

on_delete_window
on_take_focus
`Callback` objects that run with no arguments when a `WM_DELETE_WINDOW` or `WM_TAKE_FOCUS` event occurs. See wm(3tk). These are connected to nothing by default.

class teek.**Window**(**args*, ***kwargs*)
A convenient widget that represents a Ttk frame inside a toplevel.

Tk's windows like *`Toplevel`* are *not* Ttk widgets, and there are no Ttk window widgets. If you add Ttk widgets to Tk windows like *`Toplevel`* so that the widgets don't fill the entire window, your GUI looks messy on some systems, like my linux system with MATE desktop. This is why you should always create a big Ttk frame that fills the window, and then add all widgets into that frame. That's kind of painful and most people don't bother with it, but this class does that for you, so you can just create a *`Window`* and add widgets to that.

All initialization arguments are passed to *`Toplevel`*.

The *`config`* attribute combines options from the *`Frame`* and the *`Toplevel`* so that it uses *`Frame`* options whenever they are available, and *`Toplevel`* options otherwise. For example, *`Frame`* has an option named `'width'`, so `some_window.config['width']` uses that, but frames don't have a `'menu'` option, so `some_window.config['menu']` uses the toplevel's menu option.

There is no manual page for this class because this is purely a teek feature; there is no `window` widget in Tk.

See also:

`Toplevel`, *`Frame`*

toplevel
The *`Toplevel`* widget that the frame is in. The *`Window`* object itself has all the attributes and methods of the *`Frame`* inside the window, and for convenience, also many *`Toplevel`* things like *`title`*, *`withdraw()`* and *`on_delete_window`*.

destroy()
Destroys the *`toplevel`* and the frame in it.

This overrides *`Widget.destroy()`*.

2.3 Binding

Button widgets have a `'command'` option, and that is set to a callback that runs when the button is clicked. But what if you have some other widget instead of a button, and you want to do something when it's clicked? This can be useful; for example, my editor displays its logo in the about dialog, and if the logo is clicked, it shows the logo in full size.

Bindings are not limited to just clicking. You can bind to some other things as well, such as mouse movement and key presses.

2.3.1 Simple Example

This program displays a label that prints hello when it's clicked.

```
import teek

def on_click():
    print("hello")

window = teek.Window()
label = teek.Label(window, "Click me")
label.pack()
label.bind('<Button-1>', on_click)

window.on_delete_window.connect(teek.quit)
teek.run()
```

If you change `'<Button-1>'` to `'<Motion>'`, `'hello'` is printed every time you move the mouse over the label. See bind(3tk) for all possible strings you can pass to *bind()*.

Tip: If you want to bind to all widgets in `window`, you can use `window.toplevel.bind()`. See *Window* documentation for more information about `window.toplevel`.

2.3.2 Returning 'break'

Teek lets you prevent the default action of a binding from happening. For example, clicking a *Text* widget focuses the widget and moves the mouse to the location that was clicked. In this example, `on_click()` prevents this from happening by returning `'break'`:

```
import teek

window = teek.Window("Hello")

text = teek.Text(window)
text.pack()

def on_click():
    print("clicked, returning 'break'")
    return 'break'

text.bind('<Button-1>', on_click)
```

(continues on next page)

(continued from previous page)

```
window.on_delete_window.connect(teek.quit)
teek.run()
```

Note that `return 'break'` also prevents other `<Button-1>` bindings from running. See *`Callback.connect()`* for more details.

2.3.3 Event Objects

In Tcl, code like this...

```
bind $widget <Button-1> {puts "clicked at %X,%Y"}
```

...means that if the widget is clicked 10 pixels right and 20 pixels down from the top left corner of the widget's parent window, `clicked at 10,20` will be printed. Here `puts "clicked at %X,%Y"` is a string of code, and `%X` and `%Y` are replaced with the coordinates before it's ran.

This teek code does the same thing:

```
def on_click(event):
    print("clicked at {},{}".format(event.rootx, event.rooty))

widget.bind('<Button-1>', on_click, event=True)
```

Here `on_click()` gets an event object. It has attributes that describe what happened, like `rootx` and `rooty`. If you don't pass `event=True`, the callback will be called without the event object.

The `%X` and `%Y` stuff is documented in bind(3tk), but event(3tk) seems to contain more useful things. Event object attributes are named similarly as in event(3tk); for example, there is a `-rootx` option in event(3tk), and that's why teek's event objects have a `rootx` attribute.

Here is a long table of attributes that teek supports. It took me a long time to make. The list also demonstrates how limited tkinter is; only a few things in this table are supported in tkinter.

Name in event(3tk) and teek event attribute	Type in teek	Tkinter event attribute, if any	Tcl substitution bind
`above`	`int`		`%a`
`borderwidth`	`int`		`%B`
`button`	`int`	`num`	`%b`
not in event(3tk) `char` in teek	`str`	`char`	`%A`
`count`	`int`		`%c`
`data`	see *Virtual Events*		`%d`
`detail` in event(3tk) `event.data(str)` in teek	`str`		`%d`
`delta`	`int`	`delta`	`%D`
`focus`	`bool`	`focus`	`%f`
`height`	`int`	`height`	`%h`
not in event(3tk) `i_window` in teek	`int`		`%i`
`keycode`	`int`	`keycode`	`%k`
`keysym`	`str`	`keysym`	`%K`
not in event(3tk) `keysym_num` in teek	`int`	`keysym_num`	`%N`
`mode`	`str`		`%m`
`override`	`bool`		`%o`
`place`	`str`		`%p`
not in event(3tk) `property_name` in teek	`str`		`%P`

Continued on next p

Table 1 – continued from previous page

Name in event(3tk) and teek event attribute	Type in teek	Tkinter event attribute, if any	Tcl substitution bind
`root`	`int`		`%R`
`rootx`	`int`	`x_root`	`%X`
`rooty`	`int`	`y_root`	`%Y`
`sendevent`	`bool`	`send_event`	`%E`
`serial`	`int`	`serial`	`%#`
`state`	`str`	`state`	`%s`
`subwindow`	`int`		`%S`
`time`	`int`	`time`	`%t`
not in event(3tk) `type` in teek	`int`	`type`	`%T`
not in event(3tk) `widget` in teek	*`Widget`*	`widget`	`%W`
`width`	`int`	`width`	`%w`
`x`	`int`	`x`	`%x`
`y`	`int`	`y`	`%y`

Note that `%d` is used for both `detail` and `data` in Tcl, depending on the type of the event. Teek uses that internally, but it doesn't keep track of the event types for you, so you need to do `event.data(str)` if you want the `detail` string.

Note: If the value is not available, it's usually None, but the attributes whose teek type is `str` are `'??'` instead. The reason is that the Tcl substitution gets a `??` value for some reason in these cases, but `'??'` could be also a valid value of e.g. `data`, so teek doesn't try to hide it.

The "Tcl bind substitution" and "tkinter event attribute" columns are for porting Tcl code and tkinter code to teek. If you are writing a new program in teek, don't worry about them.

2.3.4 The bindings attribute

Teek uses *`Callback`* objects for most things that it runs for you. It also does that with bindings.

`teek.Widget.`**`bindings`**
: A dictionary-like object of the widget's bindings with string keys and *`Callback`* values.

  Some binding sequences are equivalent in Tk. For example, `<ButtonPress-1>`, `<Button-1>` and `<1>` all mean the same thing, and looking up those strings from a widget's `bindings` is guaranteed to give the same *`Callback`* object.

`teek.Widget.`**`bind`**(*sequence*, *func*, *, *event=False*)
: For convenience, `widget.bind(sequence, func, event=True)` does `widget.bindings[sequence].connect(func)`. Note that this does not discard old bindings, so calling this repeatedly will result in multiple functions being bound at the same time (unlike in tkinter, see *Binding* in the tkinter porting tutorial).

  If `event=True` is not given, `widget.bindings[sequence]` is connected to a new function that calls `func` with no arguments, ignoring the event object.

2.3.5 Class Bindings

Sometimes it's useful to bind things so that all instances of a teek class get bound.

`teek.Widget.`**`class_bindings`**

teek.Widget.**bind_class**(*sequence*, *func*, *, *event=False*)

These are like *bindings* and *bind()*, but for binding all instances of a class. Call `teek.Widget.bind_class()` to bind all widgets in the whole program, or e.g. `teek.Text.bind_class()` to bind all text widgets. This works both for widgets that have been already created and for widgets that will be created after the `bind_class()` call.

Note: This does not work well for classes that inherit from teek's widget classes. For example, if you have a class like this...

```
class MyText(teek.Text):
    pass
```

...then `MyText.bind_class` and `MyText.class_bindings` are no different from `teek.Text.bind_class` and `teek.Text.class_bindings`. This is because `class_bindings` and `bind_class()` use the *tk_class_name* attribute.

2.3.6 Virtual Events

Names of virtual events have << and >> instead of < and >. Here is an example:

```
>>> window = teek.Window()
>>> label = teek.Label(window)
>>> label.bind('<<Asd>>', print, event=True)    # will run print(the_event)
>>> label.event_generate('<<Asd>>')      # doctest: +ELLIPSIS
<Event: data='', serial=..., type=35>
```

You can also pass data to the virtual event:

```
>>> label.event_generate('<<Asd>>', data='toot')      # doctest: +ELLIPSIS
<Event: data='toot', serial=..., type=35>
```

If you want to actually use the data, don't do just `event.data`; that doesn't work right. Instead, use `event.data(type_spec)` where `type_spec` is a *type specifacion*. For example, `event.data([str])` retrieves the data as a list of strings.

```
>>> def callback(event):
...     print("got data string list:", event.data([str]))
...
>>> label.bind('<<ThingyThing>>', callback, event=True)
>>> label.event_generate('<<ThingyThing>>', data=['a', 'b', 'c'])  # doctest:␣
↪+ELLIPSIS
got data string list: ['a', 'b', 'c']
```

Widget.**event_generate**(*event*, ***kwargs*)

Calls `event generate` documented in event(3tk).

As usual, options are given without dashes as keyword arguments, so Tcl code like `event generate $widget <SomeEvent> -data $theData` looks like `widget.event_generate('<SomeEvent>', data=the_data)` in teek.

2.4 Dialogs

This page contains things for asking the user things like file names and colors. If you want to display a custom dialog, create a *Window*, add some stuff to it and use *wait_window()*.

Note: All functions documented on this page take a `parent` keyword argument. Use that whenever you are calling the functions from a program that has another window. This way the dialog will look like it belongs to that parent window.

2.4.1 Message Boxes

These functions call tk_messageBox(3tk). Options are passed to tk_messageBox(3tk) so that this code...

```
teek.dialog.ok_cancel("Question", "Do you want that?")
```

...does a Tcl call like this:

```
tk_messageBox -type okcancel -title "Question" -message "Do you want that?" -icon␣
↪question
```

`teek.dialog.`**`info`**(*title*, *message*, *detail=None*, ***kwargs*)
`teek.dialog.`**`warning`**(*title*, *message*, *detail=None*, ***kwargs*)
`teek.dialog.`**`error`**(*title*, *message*, *detail=None*, ***kwargs*)
Each of these functions shows a message box that has an "ok" button. The icon option is `'info'`, `'warning'` or `'error'` respectively. These functions always return `None`.

`teek.dialog.`**`ok_cancel`**(*title*, *message*, *detail=None*, ***kwargs*)
Shows a message box with "ok" and "cancel" buttons. The icon is `'question'` by default, but you can change it by passing a keyword argument, e.g. `icon='warning'`. Returns `True` if "ok" is clicked, and `False` if "cancel" is clicked.

`teek.dialog.`**`retry_cancel`**(*title*, *message*, *detail=None*, ***kwargs*)
Like *ok_cancel()*, but with a "retry" button instead of an "ok" button and `'warning'` as the default icon.

`teek.dialog.`**`yes_no`**(*title*, *message*, *detail=None*, ***kwargs*)
Shows a message box with "yes" and "no" buttons. The icon is `'question'` by default. Returns `True` for "yes" and `False` for "no".

`teek.dialog.`**`yes_no_cancel`**(*title*, *message*, *detail=None*, ***kwargs*)
Shows a message box with "yes", "no" and "cancel" buttons. The icon is `'question'` by default. Returns one of the strings `'yes'`, `'no'` or `'cancel'`.

`teek.dialog.`**`abort_retry_ignore`**(*title*, *message*, *detail=None*, ***kwargs*)
Like *yes_no_cancel()*, but with different buttons and return value strings. The icon is `'error'` by default.

2.4.2 File and Directory Dialogs

Keyword arguments work as usual. Note that paths are returned as strings of absolute paths, not e.g. `pathlib.Path` objects.

`teek.dialog.`**`open_file`**(***kwargs*)
Ask the user to choose an existing file. Returns the path.

This calls tk_getOpenFile(3tk) without `-multiple`. `None` is returned if the user cancels.

teek.dialog.**open_multiple_files**(***kwargs*)
Ask the user to choose one or more existing files. Returns a list of paths.

This calls tk_getOpenFile(3tk) with `-multiple` set to true. An empty list is returned if the user cancels.

teek.dialog.**save_file**(***kwargs*)
Ask the user to choose a path for a new file. Return the path.

This calls tk_getSaveFile(3tk), and returns `None` if the user cancels.

teek.dialog.**directory**(***kwargs*)
Asks the user to choose a directory, and return a path to it.

This calls tk_chooseDirectory(3tk), and returns `None` if the user cancels.

Note: By default, the user can choose a directory that doesn't exist yet. This behaviour is documented in tk_chooseDirectory(3tk). If you want the user to choose an existing directory, use `mustexist=True`.

2.4.3 Other Dialogs

teek.dialog.**color**(***kwargs*)
Calls tk_chooseColor(3tk).

The color selected by the user is returned, or `None` if the user cancelled the dialog.

2.5 Miscellaneous Classes

This page contains documentation of classes that represent different kinds of things.

2.5.1 Colors

class teek.**Color**(**args*)
Represents an RGB color.

There are a few ways to create color objects:

- `Color(red, green, blue)` creates a new color from an RGB value. The `red`, `green` and `blue` should be integers between 0 and 255 (inclusive).
- `Color(hex_string)` creates a color from a hexadecimal color string. For example, `Color('#ff0000')` is equivalent to `Color(0xff, 0x00, 0x00)` where `0xff` is hexadecimal notation for 255, and `0x00` is 0.
- `Color(color_name)` creates a color object from a Tk color. There is a long list of color names in colors(3tk).

Examples:

```
>>> Color(255, 255, 255)    # r, g and b are all maximum, this is white
<Color '#ffffff': red=255, green=255, blue=255>
>>> Color('white')      # 'white' is a Tk color name
<Color 'white': red=255, green=255, blue=255>
```

The string argument things are implemented by letting Tk interpret the color, so all of the ways to define colors as strings shown in Tk_GetColor(3tk) are supported.

Color objects are hashable, and they can be compared with ==:

```
>>> Color(0, 0, 255) == Color(0, 0, 255)
True
>>> Color(0, 255, 0) == Color(0, 0, 255)
False
```

Color objects are immutable. If you want to change a color, create a new Color object.

red
green
blue
These are the values passed to `Color()`.

```
>>> Color(0, 0, 255).red
0
>>> Color(0, 0, 255).green
0
>>> Color(0, 0, 255).blue
255
```

Assigning to these like `some_color.red = 255` raises an exception.

classmethod from_tcl(*color_string*)
`Color.from_tcl(color_string)` returns `Color(color_string)`.

This is just for compatibility with *type specifications*.

to_tcl()
Return this color as a Tk-compatible string.

The string is *often* a hexadecimal `'#rrggbb'` string, but not always; it can be also e.g. a color name like `'white'`. Use *red*, *green* and *blue* if you want a consistent representation.

```
>>> Color(255, 0, 0).to_tcl()
'#ff0000'
>>> Color('red').to_tcl()
'red'
>>> Color('red') == Color(255, 0, 0)
True
```

2.5.2 Callbacks

class teek.**Callback**
An object that calls functions.

Example:

```
>>> c = Callback()
>>> c.connect(print, args=["hello", "world"])
>>> c.run()    # runs print("hello", "world"), usually teek does this
hello world
>>> c.connect(print, args=["hello", "again"])
>>> c.run()
hello world
hello again
```

connect (*function*, *args=()*, *kwargs=None*)
Schedule `callback(*args, **kwargs)` to run.

If some arguments are passed to *run()*, they will appear before the *args* given here. For example:

```
>>> c = Callback()
>>> c.connect(print, args=['hello'], kwargs={'sep': '-'})
>>> c.run(1, 2)      # print(1, 2, 'hello', sep='-')
1-2-hello
```

The callback may return `None` or `'break'`. In the above example, `print` returned `None`. If the callback returns `'break'`, two things are done differently:

1. No more connected callbacks will be ran.
2. *run()* returns `'break'`, so that the code that called *run()* knows that one of the callbacks returned `'break'`. This is used in *bindings*.

disconnect (*function*)
Undo a *connect()* call.

Note that this method doesn't do anything to the *args* and *kwargs* passed to *connect()*, so when disconnecting a function connected multiple times with different arguments, only the first connection is undone.

```
>>> c = Callback()
>>> c.connect(print, ["hello"])
>>> c.connect(print, ["hello", "again"])
>>> c.run()
hello
hello again
>>> c.disconnect(print)
>>> c.run()
hello
```

run (**args*)
Run the connected callbacks.

If a callback returns `'break'`, this returns `'break'` too without running more callbacks. If all callbacks run without returning `'break'`, this returns `None`. If a callback raises an exception, a traceback is printed and `None` is returned.

2.5.3 Fonts

There are different kinds of fonts in Tk:

- **Named fonts** are mutable; if you set the font of a widget to a named font and you then change that named font (e.g. make it bigger), the widget's font will change as well.
- **Anonymous fonts** don't work like that, but they are handy if you don't want to create a font object just to set the font of a widget.

For example, if you have *Label*...

```
>>> window = teek.Window()
>>> label = teek.Label(window, "Hello World")
>>> label.pack()
```

...and you want to make its text bigger, you can do this:

```
>>> label.config['font'] = ('Helvetica', 20)
```

This form is the teek equivalent of the alternative `[3]` in the `FONT DESCRIPTIONS` part of font(3tk). All of the other font descriptions work as well.

If you then get the font of the label, you get a *`teek.Font`* object:

```
>>> label.config['font']
Font('Helvetica 20')
```

With a named font, the code looks like this:

```
>>> named_font = teek.NamedFont(family='Helvetica', size=20)
>>> label.config['font'] = named_font
>>> named_font.size = 50    # even bigger! label will use this new size automatically
```

Of course, *`Font`* and *`NamedFont`* objects can also be set to `label.config['font']`.

class `teek.`**Font**(*font_description*)

Represents an anonymous font.

Creating a *`Font`* object with a valid font name as an argument returns a *`NamedFont`* object. For example:

```
>>> teek.Font('Helvetica 12')  # not a font name
Font('Helvetica 12')
>>> teek.Font('TkFixedFont')   # special font name for default monospace font
NamedFont('TkFixedFont')
>>> teek.NamedFont('TkFixedFont')    # does the same thing
NamedFont('TkFixedFont')
```

family
size
weight
slant
underline
overstrike

See font(3tk) for a description of each attribute. `size` is an integer, `underline` and `overstrike` are bools, and other attributes are strings. You can set values to these attributes only with *`NamedFont`*.

The values of these attributes are looked up with `font actual` in font(3tk), so they might differ from the values passed to `Font()`. For example, the `'Helvetica'` family can meany any Helvetica-like font, so this line of code gives different values platform-specifically:

```
>>> teek.Font('Helvetica 12').family     # doctest: +SKIP
'Nimbus Sans L'
```

classmethod families(*, *allow_at_prefix=False*)

Returns a list of font families as strings.

On Windows, some font families start with `'@'`. I don't know what those families are and how they might be useful, but most of the time tkinter users (including me) ignore those, so this method ignores them by default. Pass `allow_at_prefix=True` to get a list that includes the `'@'` fonts.

classmethod from_tcl(*font_description*)

`Font.from_tcl(font_description)` returns `Font(font_description)`.

This is just for compatibility with *type specifications*.

measure(*text*)
Calls `font measure` documented in font(3tk), and returns an integer.

metrics()
Calls `font metrics` documented in font(3tk), and returns a dictionary that has at least the following keys:

- The values of `'ascent'`, `'descent'` and `'linespace'` are integers.
- The value of `'fixed'` is True or False.

to_named_font()
Returns a *NamedFont* object created from this font.

If this font is already a *NamedFont*, a copy of it is created and returned.

to_tcl()
Returns the font description passed to `Font(font_description)`.

class teek.**NamedFont**(*name=None*, ***kwargs*)
A font that has a name in Tcl.

NamedFont is a subclass of *Font*; that is, all NamedFonts are Fonts, but not all Fonts are NamedFonts:

```
>>> isinstance(teek.NamedFont('toot'), teek.Font)
True
>>> isinstance(teek.Font('Helvetica 12'), teek.NamedFont)
False
```

If `name` is not given, Tk will choose a font name that is not in use yet. If `name` is given, it can be a name of an existing font, but if a font with the given name doesn't exist, it'll be created instead.

The `kwargs` are values for `family`, `size`, `weight`, `slant`, `underline` and `overstrike` attributes. For example, this...

```
shouting_font = teek.NamedFont(size=30, weight='bold')
```

...does the same thing as this:

```
shouting_font = teek.NamedFont()
shouting_font.size = 30
shouting_font.weight = 'bold'
```

delete()
Calls `font delete`.

The font object is useless after this, and most things will raise *TclError*.

classmethod get_all_named_fonts()
Returns a list of all *NamedFont* objects.

2.5.4 Tcl Variable Objects

class teek.**TclVariable**(*, *name=None*)
Represents a global Tcl variable.

In Tcl, it's possible to e.g. run code when the value of a variable changes, or wait until the variable is set. Python's variables can't do things like that, so Tcl variables are represented as *TclVariable* objects in Python. If you want to set the value of the variable object, `variable_object = new_value` doesn't work

because that only sets a Python variable, and you need `variable_object.set(new_value)` instead. Similarly, `variable_object.get()` returns the value of the Tcl variable.

The *`TclVariable`* class is useless by itself. Usable variable classes are subclasses of it that override *`type_spec`*.

Use `SomeUsableTclVarSubclass(name='asd')` to create a variable object that represents a Tcl variable named `asd`, or `SomeUsableTclVarSubclass()` to let teek choose a variable name for you.

`type_spec`
This class attribute should be set to a *type specification* of what *`get()`* returns.

`classmethod from_tcl`(*varname*)
Creates a variable object from a name string.

See *Type Specifications* for details.

`get`()
Returns the value of the variable.

`set`(*new_value*)
Sets the value of the variable.

The value does not need to be of the variable's type; it can be anything that can be *converted to tcl*.

`to_tcl`()
Returns the variable name as a string.

`wait`()
Waits for this variable to be modified.

The GUI remains responsive during the waiting. See `tkwait variable` in tkwait(3tk) for details.

`write_trace`
A *`Callback`* that runs when the value of the variable changes.

The connected functions will be called with one argument, the variable object. This is implemented with `trace add variable`, documented in trace(3tcl).

`class` `teek.`**`StringVar`**
`class` `teek.`**`IntVar`**
`class` `teek.`**`FloatVar`**
`class` `teek.`**`BooleanVar`**
Handy *`TclVariable`* subclasses for variables with `str`, `int`, `float` and `bool` values, respectively.

2.5.5 Images

`class` `teek.`**`Image`**(***kwargs*)
Represents a Tk photo image.

If you want to display an image to the user, use *`Label`* with its `image` option. See examples/image.py.

Image objects are wrappers for things documented in image(3tk) and photo(3tk). They are mutable, so you can e.g. set a label's image to an image object and then later change that image object; the label will update automatically.

Note: PNG support was added in Tk 8.6. Use GIF images if you want backwards compatibility with Tk 8.5.

If you want to create a program that can read as many different kinds of images as possible, use *`teek.extras.image_loader`*.

Creating a new `Image` object with `Image(...)` calls `image create photo` followed by the options in Tcl. See image(3tk) for details.

Keyword arguments are passed as options to photo(3tk) as usual, except that if a `data` keyword argument is given, it should be a `bytes` object of data that came from e.g. an image file opened with `'rb'`; it will be automatically converted to base64.

Image objects can be compared with ==, and they compare equal if they represent the same Tk image; that is, `image1 == image2` returns `image1.to_tcl() == image2.to_tcl()`. Image objects are also hashable.

config
Similar to *the widget config attribute*.

blank()
See `imageName blank` in photo(3tk).

copy(***kwargs*)
Create a new image with the same content as this image so that changing the new image doesn't change this image.

This creates a new image and then calls *copy_from()*, so that this...

```
image2 = image1.copy()
```

...does the same thing as this:

```
image2 = teek.Image()
image2.copy_from(image1)
```

Keyword arguments passed to `image1.copy()` are passed to `image2.copy_from()`. This means that it is possible to do some things with both *copy()* and *copy_from()*, but *copy()* is consistent with e.g. `list.copy()` and `dict.copy()`.

copy_from(*source_image*, ***kwargs*)
See `imageName copy sourceImage` documented in photo(3tk).

Options are passed as usual, except that `from=something` is invalid syntax in Python, so this method supports `from_=something` instead. If you do `image1.copy_from(image2)`, the `imageName` in photo(3tk) means `image1`, and `sourceImage` means `image2`.

delete()
Calls `image delete` documented in image(3tk).

The image object is useless after this, and most things will raise *TclError*.

classmethod from_tcl(*name*)
Create a new image object from the name of a Tk image.

See *Type Specifications* for details.

get(*x*, *y*)
Returns the *Color* of the pixel at (x,y).

classmethod get_all_images()
Return all existing images as a list of *Image* objects.

get_bytes(*format_*, ***kwargs*)
Like *write()*, but returns the data as a `bytes` object instead of writing it to a file.

The `format_` argument can be any string that is compatible with the `-format` option of `imageName write` documented in photo(3tk). All keyword arguments are same as for *write()*.

height
See *width*.

in_use()
True if any widget uses this image, or False if not.

This calls `image inuse` documented in image(3tk).

read(*filename*, ***kwargs*)
See `imageName read filename` in photo(3tk).

redither()
See `imageName redither` in photo(3tk).

to_tcl()
Returns the Tk name of the image as a string.

transparency_get(*x*, *y*)
Check if the pixel at (x,y) is transparent, and return a bool.

The *x* and *y* are pixels, as integers. See `imageName transparency get` in photo(3tk).

transparency_set(*x*, *y*, *is_transparent*)
Make the pixel at (x,y) transparent or not transparent.

See `imageName transparency set` in photo(3tk) and *transparency_get()*.

width
The current width of the image as pixels.

Note that `image.width` is different from `image.config['width']`; `image.width` changes if the image's size changes, but `image.config['width']` often represents the width that the image had when it was first created. **tl;dr:** Usually it's best to use `image.width` instead of `image.config['width']`.

write(*filename*, ***kwargs*)
See `imageName write` in photo(3tk).

See also:

Use *get_bytes()* if you don't want to create a file.

2.5.6 Screen Distance Objects

class `teek.`**ScreenDistance**(*value*)
Represents a Tk screen distance.

If you don't know or care what screen distances are, use the *pixels* attribute. The `value` can be an integer or float of pixels or a string that Tk_GetPixels(3tk) accepts; for example, `123` or `'2i'`.

`ScreenDistance` objects are hashable, and they can be compared with each other:

```
>>> funny_dict = {ScreenDistance(1): 'lol'}
>>> funny_dict[ScreenDistance(1)]
'lol'
>>> ScreenDistance('1c') == ScreenDistance('1i')
False
>>> ScreenDistance('1c') < ScreenDistance('1i')
True
```

pixels
The number of pixels that this screen distance represents as an int.

This is implemented with `winfo pixels`, documented in winfo(3tk).

fpixels
The number of pixels that this screen distance represents as a float.

This is implemented with `winfo fpixels`, documented in winfo(3tk).

classmethod from_tcl(*value_string*)
Creates a screen distance object from a Tk screen distance string.

See *Type Specifications* for details.

to_tcl()
Return the `value` as a string.

2.6 Platform Information

This page documents things that can tell you which platform your program is running on.

`teek.`**TCL_VERSION**
`teek.`**TK_VERSION**
These can be used for checking the versions of the Tcl interpreter and its Tk library that teek is using. These are two-tuples of integers, and you can compare integer tuples nicely, so you can do e.g. this:

```
if teek.TK_VERSION >= (8, 6):
    # use a feature new in Tk 8.6
else:
    # show an error message or do things without the new feature
```

Teek refuses to run if Tcl or Tk is older than 8.5, so you can use all features new in Tcl/Tk 8.5 freely.

Note: The manual page links in this tutorial like label(3tk) always point to the latest manual pages, which are for Tcl/Tk 8.6 at the time of writing this.

`teek.`**windowingsystem**()
This function returns `'win32'`, `'aqua'` or `'x11'`. Use it instead of `platform.system()` when you have platform-specific teek code. For example, it's possible to run X11 on a Mac, in which case `platform.system()` returns `'Darwin'` and this function returns `'x11'`. If you have code that should even then behave like it would normally behave on a Mac, use `platform.system()`.

The Tk documentation for this function is `tk windowingsystem` in tk(3tk).

2.7 Extras

Big tkinter projects often have their own implementations of some commonly needed things that tkinter doesn't come with. The `teek.extras` module contains a collection of them.

To use the extras, `import teek` is not enough; that doesn't import any extras. This is good because most teek programs don't need the extras, and for them, `import teek` may run a little bit faster if it doesn't import the extras. Instead, you typically need to do something like this to use the extras:

```
import teek
from teek.extras import tooltips

# now some code that uses tooltips.set_tooltip
```

2.7.1 tooltips

This module contains a simple tooltip implementation with teek. There is example code in examples/tooltip.py.

If you have read some of IDLE's source code (if you haven't, that's good; IDLE's source code is ugly), you might be wondering what this thing has to do with `idlelib/tooltip.py`. Don't worry, I didn't copy/paste any code from idlelib and I didn't read idlelib while I wrote the tooltip code! Idlelib is awful and I don't want to use anything from it in my projects.

`teek.extras.tooltips.`**`set_tooltip`**(*widget*, *text*)

Create tooltips for a widget.

After calling `set_tooltip(some_widget, "hello")`, "hello" will be displayed in a small window when the user moves the mouse over the widget and waits for a small period of time. Do `set_tooltip(some_widget, None)` to get rid of a tooltip.

2.7.2 cross_platform

Most teek things work the same on most platforms, but not everything does. For example, binding `<Tab>` works everywhere, but binding `<Shift-Tab>` doesn't work on Linux and you need a different binding instead. This extra module contains utilities for dealing with things like that.

`teek.extras.cross_platform.`**`bind_tab_key`**(*widget*, *callback*, ***bind_kwargs*)

A cross-platform way to bind Tab and Shift+Tab.

Use this function like this:

```
from teek.extras import cross_platform

def on_tab(shifted):
    if shifted:
        print("Shift+Tab was pressed")
    else:
        print("Tab was pressed")

cross_platform.bind_tab_key(some_widget, on_tab)
```

Binding `'<Tab>'` works on all systems I've tried it on, but if you also want to bind tab presses where the shift key is held down, use this function instead.

This function can also take any of the keyword arguments that *`teek.Widget.bind()`* takes. If you pass `event=True`, the callback will be called like `callback(shifted, event)`; that is, the `shifted` bool is the first argument, and the event object is the second.

2.7.3 more_dialogs

This is useful when *`teek.dialog`* is not enough.

All of the functions take these arguments:

- `title` will be the title of the dialog.

- `text` will be displayed in a label above the text entry or spinbox.
- `initial_value` will be added to the entry or spinbox before the user changes it.
- `parent` is a window widget that the dialog will be displayed on top of.

`teek.extras.more_dialogs.`**`ask_string`**(*title*, *text*, ***, *validator=<class 'str'>*, *initial_value=''*, *parent=None*)

Displays a dialog that contains a *`teek.Entry`* widget.

The `validator` should be a function that takes a string as an argument, and returns something useful (see below). By default, it returns the string unchanged, which is useful for asking a string from the user. If the validator raises `ValueError`, the OK button of the dialog is disabled to tell the user that the value they entered is invalid. Then the user needs to enter a valid value or cancel the dialog.

This returns whatever `validator` returned, or `None` if the dialog was canceled.

`teek.extras.more_dialogs.`**`ask_integer`**(*title*, *text*, *allowed_values*, ***, *initial_value=None*, *parent=None*)

Displays a dialog that contains a *`teek.Spinbox`* widget.

`allowed_values` can be a sequence of acceptable integers or a `range`. If `initial_value` is given, it must be in `allowed_values`. If it's not, `allowed_values[0]` is used.

This returns an integer in `allowed_values`, or `None` if the user cancels.

2.7.4 links

With this extra, you can insert web browser style links to *`Text`* widgets. This is based on this tutorial that is almost as old as I am, but it's still usable.

See examples/links.py for example code.

`teek.extras.links.`**`add_url_link`**(*textwidget*, *url*, *start*, *end*)

Make some of the text in the textwidget to be clickable so that clicking it will open `url`.

The text between the *text indexes* `start` and `end` becomes clickable, and rest of the text is not touched.

Do this if you want to insert some text and make it a link immediately:

```
from teek.extras import links

...

old_end = textwidget.end    # adding text to end changes textwidget.end
textwidget.insert(textwidget.end, 'Click me')
links.add_url_link(textwidget, 'https://example.com/', old_end, textwidget.end)
```

This function uses `webbrowser.open()` for opening `url`.

`teek.extras.links.`**`add_function_link`**(*textwidget*, *function*, *start*, *end*)

Like *`add_url_link()`*, but calls a function instead of opening a URL in a web browser.

2.7.5 image_loader

Note: This extra has dependencies that don't come with teek when you install it with `pip install teek` or similar, because I want teek to be light-weight and I don't want to bring in lots of dependencies with it. Run `pip install teek[image_loader]` to install the things you need for using this extra.

There's also `image_loader_dummy`, which is like `image_loader` except that it just creates *`teek.Image()`* objects, and doesn't support anything that plain *`teek.Image()`* doesn't support. It's meant to be used like this:

```
try:
    from teek.extras import image_loader
except ImportError:
    from teek.extras import image_loader_dummy as image_loader
```

This extra lets you create *`teek.Image`* objects of images that Tk itself doesn't support. It uses other Python libraries like PIL and svglib to do that, and you can just tell it to load an image and let it use whatever libraries are needed.

2.7.6 soup

This extra contains code that views HTML to text widgets. The HTML is taken as BeautifulSoup elements, so you need to have it installed to use this module. Don't get too excited though – this is not something that's intended to be used for creating a web browser or something, because this thing doesn't even support JavaScript or CSS! It's meant to be used e.g. when you want to display some markup to the user, and you know a library that can convert it to simple HTML.

If the HTML contains images that are not GIF images, make sure to install *`image_loader`*. The `soup` extra will use it if it's installed.

Only these HTML elements are supported by default (but you can subclass *`SoupViewer`* and add support for more elements, see *`SoupViewer.add_soup()`*):

- `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` and `<h6>`
- `<pre>` and `<code>`
- `<br>`
- `<b>`, `<i>`, `<em>` and `<strong>`
- `<p>`
- `<ul>`, `<ol>` and `<li>`
- `<a>`
- `<img>`

class `teek.extras.soup.`**SoupViewer**(*textwidget*, *threads=True*)

Displays BeautifulSoup HTML elements in a text widget.

Note: If the soup contains `<img>` tags, the images are read or downloaded automatically by default. Subclass *`SoupViewer`* and override *`download()`* if you don't want that.

Images are loaded in threads, so make sure to use *`teek.init_threads()`*. Alternatively, you can pass `threads=False`, and the images won't be loaded at all.

---

widget

The *`teek.Text`* widget that everything is added to.

add_soup(*element*)

Render a BeautifulSoup4 HTML element.

The text, images, or whatever the element represents are added to the end of the text widget.

This method looks for methods whose names are `handle_` followed by the name of a HTML tag; for example, `handle_h1()` or `handle_p()`. Those methods run when an element with the corresponding tag name is added. You can subclass *`SoupViewer`* and create more of these methods to handle more different kinds of tags. There are two things that the methods can do:

1. The method can return None to indicate that *`add_soup()`* shouldn't do anything with the content of the element.

```
def handle_pre(self, pre):
    self.widget.insert(self.widget.end, pre.text.rstrip() + '\n\n')
```

2. The method can be decorated with `contextlib.contextmanager()`. When it yields, *`add_soup()`* will loop over the element and call itself recursively with each subelement.

```
@contextlib.contextmanager
def handle_ul(self, ul):
    for li in ul:
        if li.name == 'li':
            # '\N{bullet} ' creates a Unicode black circle character
            li.insert(0, '\N{bullet} ')
    yield     # the content of the ul is added here
    self.widget.insert(self.widget.end, '\n')
```

In either case, *`add_soup()`* adds a *textwidget tag* as explained in *`create_tags()`*.

`create_tags`()

Adds *text tags* to the *`widget`* for displaying the soup elements.

This is not called automatically; you should call this before actually using the `SoupViewer`.

Each text tag is named with `'soup-'` followed by the name of the corresponding HTML tag, such as `'soup-p'` or `'soup-pre'`. If you are not happy with what this method does, you can change the text tags after calling it.

`download`(*url*)

Downloads the content of the URL, and returns it as bytes.

This method is called whenever the soup contains an `<img>` or something else that has to be read from a file or downloaded. If it raises an exception, the `alt` of the `<img>` will be displayed instead of the actual image, if there is an `alt`. The `alt` is also displayed while this method is running.

By default, this uses `urllib.request.urlopen()`. You can override this if `urllib` is doing something dumb or you want to control which things can be downloaded.

Usually this is called from some other thread than the main thread.

`stop_loading`(*cleanup=True*)

Tell currently running threads to do nothing to the *`widget`*.

Things like `<img>` elements are loaded with threads, so they might add something to the text widget several seconds after the *`add_soup()`* call.

If `cleanup` is `True`, this method also e.g. deletes already loaded images, because then it assumes that they are not needed anymore. This means that if you don't pass `cleanup=False`, you should clear the text widget after calling this method.

This is automatically called with `cleanup=True` when the *`widget`* is destroyed.

CHAPTER 3

Understanding teek

Things documented here are useful if you want to know how stuff works or you want to do some advanced tricks with teek. I recommend reading these things if you want to help me with developing teek.

3.1 Event Loop

Tk is event-based. When you click a *Button*, a click event is generated, and Tk processes it. Usually that involves making the button look like it's pressed down, and maybe calling a callback function that you have told the button to run.

The **event loop** works essentially like this pseudo code:

```
while True:
    handle_an_event()
    if there_are_no_more_events_because_we_handled_all_of_them:
        wait_for_more_events()
```

These functions can be used for working with the event loop:

`teek.run()`
: Runs the event loop until *quit()* is called.

`teek.quit()`
: Stop the event loop and destroy all widgets.

  This function calls `destroy .` in Tcl, and that's documented in destroy(3tk). Note that this function does not tell Python to quit; only teek quits, so you can do this:

  ```
  import teek

  window = teek.Window()
  teek.Button(window, "Quit", teek.quit).pack()
  teek.run()
  print("Still alive")
  ```

If you click the button, it interrupts `teek.run()` and the print runs.

`teek.`**`before_quit`**
`quit()` runs this callback with no arguments before it does anything else. This means that when this callback runs, widgets have not been destroyed yet, but they will be destroyed soon.

`teek.`**`after_quit`**
`quit()` runs this callback when it has done everything else successfully.

`teek.`**`update`**(*, *idletasks_only=False*)
Handles all pending events, and returns when they are all handled.

See update(3tcl) for details. If `idletasks_only=True` is given, this calls `update idletasks`; otherwise, this calls `update` with no arguments.

3.2 Tcl Calls

Note: This section assumes that you know Tcl. You may have learned some of it while using teek, but something like Learn Tcl in Y Minutes might be useful for you.

Teek does most things by calling commands in Tcl. You can also call Tcl commands yourself, which is useful if you want to do something that can be done in Tcl, but there is no other way to do it in teek.

There are two functions for doing this:

`teek.`**`tcl_eval`**(*returntype*, *code*)
Run a string of Tcl code.

```
>>> teek.tcl_eval(None, 'proc add {a b} { return [expr $a + $b] }')
>>> teek.tcl_eval(int, 'add 1 2')
3
>>> teek.tcl_call(int, 'add', 1, 2)      # usually this is better, see below
3
```

`teek.`**`tcl_call`**(*returntype*, *command*, **arguments*)
Call a Tcl command.

The arguments are passed correctly, even if they contain spaces:

```
>>> teek.tcl_eval(None, 'puts "hello world thing"')  # 1 arguments to puts
↪# doctest: +SKIP
hello world thing
>>> message = 'hello world thing'
>>> teek.tcl_eval(None, 'puts %s' % message)  # 3 args to puts, tcl error
Traceback (most recent call last):
    ...
teek.TclError: wrong # args: should be "puts ?-nonewline? ?channelId? string"
>>> teek.tcl_call(None, 'puts', message)    # 1 arg to puts        # doctest: +SKIP
hello world thing
```

Both of these functions are ran so that they have access to Tcl's global variables, and if they create more variables, they will also be global.

The `None` means that the return value is ignored, and `int` means that it's converted to a Python integer. There are more details about these conversions below.

Tcl errors always raise the same Python exception:

exception teek.**TclError**
This is raised when a Tcl command fails.

3.2.1 Data Types

Everything is a string in Tcl. Teek converts Python objects to strings and strings to Python objects for you, but you need to tell teek what types of values you want to get. This section describes how.

Python to Tcl conversion

Arguments passed to *tcl_call()* are handled like this:

- Strings are passed to Tcl as is.
- If the argument is None, an empty string is passed to Tcl because Tcl uses an empty string in many places where Python uses None.
- If the argument is a dictionary-like object (more precisely, collections.abc.Mapping), it is turned into a list of pairs. This is because `{'a': 'b', 'c': 'd'}` and `['a', 'b', 'c', 'd']` are represented the same way in Tcl.
- `True` and `False` are converted to `1` and `0`, respectively.
- Integers, floats and other real numbers (numbers.Real) are converted to strings with `str()`.
- If the value has a `to_tcl()` method, it's called with no arguments. It should return a string that will be passed to Tcl.
- Anything else is treated as an iterable. Every element of the iterable is converted as described here, and the result is a Tcl list.

Conversions should raise `ValueError` or *TclError* when they fail.

Type Specifications

Teek also has **type specifications** for converting from a Tcl object to a Python object. Here is a list of valid type specifications:

- `str` (that is, literally `str`, not e.g. `'hello'`) means that a string is returned.
- `None` means that the value will be ignored entirely, and the Python value is always None.
- `bool` means that the value is treated as a Tcl boolean. All valid Tcl booleans specified in Tcl_GetBoolean(3tcl) are supported.
- `int`, `float` or any other subclass of `numbers.Real` means that the value will be converted to that class by first converting to string as if `str` was used, and then calling the class with the stringed value as an argument. However, if the stringed value is the empty string, None is returned and the class isn't called.
- If the type specification is a class with a `from_tcl()` classmethod, that will be called with one argument, the value converted to a string. If the stringed value is an empty string, None is returned and `from_tcl()` is not called.

The type specifications can be also combined in the following ways. These examples use `str`, `int` and `float`, but all other valid specifications work as well. The return types can be nested arbitrarily; for example, `[(int, float)]` means a value like `[(12, 3.4), (56, 7.8)]`.

- `[str]` means a list of strings, of any length.

- `(str, int)` means a tuple of a string followed by an integer. This allows you to create a sequence with different kinds of items in it. For example, `(str, str, str)` is like `[str]` except that it also makes sure that the length of the result is 3, and returns a tuple instead of a list.
- `{'a': int, 'b': float}` means a dictionary with string keys. If the Tcl dictionary happens to have a key named `a` or `b`, it is converted to `int` or `float` respectively; other keys will be strings. This means that `{}` is a dictionary with all keys as strings and values as integers. There is no way to work with dictionaries that have non-string keys.

Examples:

```
>>> teek.tcl_call([str], 'list', 'a', 'b', 'c')
['a', 'b', 'c']
>>> teek.tcl_call((str, int, float), 'list', 'hello', '3', '3.14')
('hello', 3, 3.14)
>>> teek.tcl_call([bool], 'list', 'yes', 'ye', 'true', 't', 'on', '1')
[True, True, True, True, True, True]
>>> teek.tcl_call({}, 'dict', 'create', 'a', 1, 'b', 2)  # doctest: +SKIP
{'a': '1', 'b': '2'}
>>> teek.tcl_call([str], 'list', 123, 3.14, None, 'hello')
['123', '3.14', '', 'hello']
```

Creating Tcl Commands

It's possible to create Tcl commands that Tcl code can call. For example, when a button is clicked, Tcl invokes a command that the *Button* class created with *create_command()*.

`teek.`**`create_command`**(*func*, *arg_type_specs=()*, *, *extra_args_type=None*)

Create a Tcl command that calls `func`.

Here is a simple example:

```
>>> tcl_print = teek.create_command(print, [str])  # calls print(a_string)
>>> tcl_print       # doctest: +SKIP
'teek_command_1'
>>> teek.tcl_call(None, tcl_print, 'hello world')
hello world
>>> teek.tcl_eval(None, '%s "hello world"' % tcl_print)
hello world
```

Created commands should be deleted with *delete_command()* when they are no longer needed.

The function will take `len(arg_type_specs)` arguments, and the arguments are converted to Python objects using `arg_type_specs`. The `arg_type_specs` must be a sequence of *type specifications*.

If `extra_args_type` is given, the function can also take more than `len(arg_type_specs)` arguments, and the type of each extra argument will be *extra_args_type*. For example:

```
>>> def func(a, b, *args):
...     print(a - b)
...     for arg in args:
...         print(arg)
...
>>> command = teek.create_command(func, [int, int], extra_args_type=str)
>>> teek.tcl_call(None, command, 123, 23, 'asd', 'toot', 'boom boom')
100
asd
```

(continues on next page)

(continued from previous page)

```
toot
boom boom
```

The return value from the Python function is *converted to a string for Tcl*.

If the function raises an exception, a traceback will be printed. However, the Tcl command returns an empty string on errors and does *not* raise an error in Tcl. Be sure to return a non-empty value on success if you want to do error handling in Tcl code.

teek.**delete_command**(*name*)
Delete a Tcl command by name.

You can delete commands returned from `create_command()` to avoid memory leaks.

Python Module Index

t

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

X

Y