

# 13 GUIs and Executables

March 23, 2022

## 1 Module 13: GUIs and Executables

March 25, 2022

Last time we dived deeper into objects and object-oriented programming (OOP) in Python. We saw how to create own classes, discuss the concept of inheritance, and had a quick look at UML class diagrams. Furthermore, we briefly talked about higher-order functions, which exploit that in Python also functions are objects.

Today we take a look at some additional practical topics: Building graphical user interfaces (GUIs) and executables with Python, which can be very useful for making functionality for third parties, especially when they can/must/should not deal with the code directly.

Next time we will discuss how to implement parallel behaviour in Python.

### 1.1 Graphical User Interfaces with Tkinter

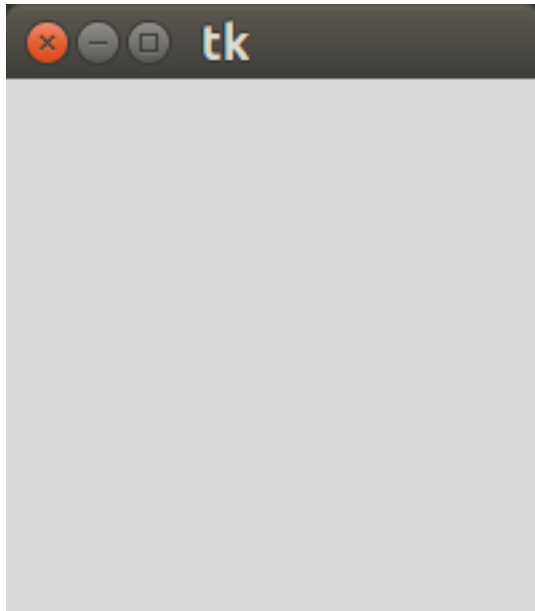
There is a large number of frameworks and toolkits available for building graphical user interfaces (GUIs) in Python (see e.g. the list at <https://wiki.python.org/moin/GuiProgramming>). We will focus here on how to create GUIs with Tkinter (see the official reference at <https://docs.python.org/3/library/tkinter.html>), which is the fairly easy-to-use standard GUI framework included in Python. The following introduction to Tkinter is largely based on the very elaborate “Thinking Tkinter” tutorial available online at <http://thinkingtkinter.sourceforge.net/>.

The simplest possible Tkinter program is probably the following:

```
[6]: import tkinter as tk

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

First the Tkinter library is imported under the name `tk` for easier reference. Then an instance of the class `Tkinter.Tk` is created, which creates a basic window object. This `root` object is the the highest-level GUI component in any Tkinter application, often also referred to as the “oplevel window”. Finally, the `mainloop` method of the root object is executed. As the name suggests, it starts the main loop of the application window. This loop runs continuously, waiting for events, handling them when they occur, and only stopping when the window is closed.



Obviously, what needs to happen from here is to add further components to the root window, and implement the functionality to handle events that occur from interaction of a user with the interface. We cannot cover all possibilities in the scope of this lecture, of course, but the following examples should give you an idea how it works.

Two kinds of GUI components are distinguished in Tkinter: containers and widgets. *Widgets* are all the things that are (usually) visible and do things, such as text fields, drop-down lists, buttons, etc. *Containers* are components that, well, contain other components, especially widgets. The most frequently used container class is **Frame**.

Here is a list of basic *widgets*: <https://tkdocs.com/tutorial/widgets.html>

The following example shows how to add a container and a “Say hello!” button to the empty window from above:

```
[13]: root = tk.Tk()

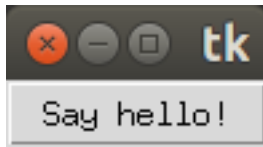
      container = tk.Frame(root)
      container.pack()

      hwbutton = tk.Button(container)
      hwbutton["text"] = "Say hello!"
      hwbutton.pack()

      root.mainloop()
```

We create a **root** object as before. Then we add a frame **container** to the base window. This establishes a logical relationship between the **container** and **root**. Furthermore, the **pack** method needs to be called to invoke a “geometry manager” and establish a visual relationship between the object and its parent, to actually make it visible. Similarly, we add a button to the container, set its text and **pack** it. Finally, the main loop of the application is started. We get a window with a button that we can click, but nothing else happens, simply because we have not defined yet what

should happen (but we will get to that).



When GUI applications get larger, it is usually advisable to follow the object-oriented programming style rather than the procedure-oriented one, and organize the code in classes. For the example from above, that could look as follows:

```
[21]: class HelloWorldApp:

    def __init__(self, parent):
        self.container = tk.Frame(parent)
        self.container.pack()
        self.hwbutton = tk.Button(self.container)
        self.hwbutton["text"] = "Say hello!"
        self.hwbutton.pack()

root = tk.Tk()
hwapp = HelloWorldApp(root)
root.mainloop()
```

Adding additional widgets to the application can be done in the same way as adding the button. The following example shows that instead of configuring widgets by using their dictionaries, this can also be done with the `configure` method or directly during their instantiation:

```
[22]: class HelloWorldApp:

    def __init__(self, parent):
        self.container = tk.Frame(parent)
        self.container.pack()
        self.hwbutton = tk.Button(self.container)
        self.hwbutton["text"] = "Say hello!"
        self.hwbutton.pack()
        self.hwtext = tk.Label(self.container)
        self.hwtext.configure(text="", background="white")
        self.hwtext.pack()
        self.gbbbutton = tk.Button(self.container, text="Goodbye!", \
                                   background="red")
        self.gbbbutton.pack()

root = tk.Tk()
hwapp = HelloWorldApp(root)
root.mainloop()
```



Note: If you are on Mac OS X, you might have to run the following code instead to see the button in red:

```
[ ]: # importing tkmacosx
import tkmacosx as tkmac

class HelloWorldApp:

    def __init__(self, parent):
        self.container = tk.Frame(parent)
        self.container.pack()
        self.hwbutton = tk.Button(self.container)
        self.hwbutton["text"] = "Say hello!"
        self.hwbutton.pack()
        self.hwtext = tk.Label(self.container)
        self.hwtext.configure(text="", background="white")
        self.hwtext.pack()
        # using Button from tkmacosx
        self.gbbbutton = tkmac.Button(self.container, text="Goodbye!", \
                                      background="red")

        self.gbbbutton.pack()

root = tk.Tk()
hwapp = HelloWorldApp(root)
root.mainloop()
```

The standard way of placing widgets within the container is on top of each other. That is because the default value of the `side` parameter of the `pack` method is in fact `top`. By using “bottom”, “left” or “right” alternatively, the orientation can be changed. To avoid unpredictable behavior when e.g. resizing the application window, it is advisable to use the same orientation for all widgets in a container. Change the code above to use `pack` always with parameter `side="left"` and see what happens.

If we would like, for example, to place the text field above the two buttons, we can easily do that by using two containers (placed on top of each other), of which one contains the text field and the other the two buttons (next to each other):

```
[51]: class HelloWorldApp:

    def __init__(self, parent):
        self.container1 = tk.Frame(parent)
```

```

self.container1.pack()
self.hwtext = tk.Label(self.container1)
self.hwtext.configure(text="", background="white")
self.hwtext.pack(side="left")
self.container2 = tk.Frame(parent)
self.container2.pack()
self.hwbutton = tk.Button(self.container2)
self.hwbutton["text"] = "Say hello!"
self.hwbutton.pack(side="left")
self.gbbutton = tk.Button(self.container2, text="Goodbye!", \
                           background="red")
self.gbbutton.pack(side="left")

root = tk.Tk()
hwapp = HelloWorldApp(root)
root.mainloop()

```



So far for creating a tk application window and adding and arranging GUI elements. Of course we also want the buttons to actually do something when we click on them. To achieve this, we need to do two things: 1) write event handler routines that do the intended things, and 2) bind these routines to the respective widgets and events.

For example, if we want a click on the “Say hello!” button to cause the text “Hello World!” to appear in the text box, and a click on the “Goodbye!” button to cause the window to close, we could define the following two methods in our application class:

```

def hwbuttonClick(self,event):
    self.hwtext.configure(text="Hello World!")

def gbbuttonClick(self,event):
    self.parent.destroy()

```

The first method simply changes the text in the text field, the second one calls the **destroy** method of the root object and thus closes the window. Furthermore, we need to register the methods at the respective buttons with the **bind** method. The first parameter of **bind** is the event that we want to handle (a click with the left mouse button is called "<Button-1>") and the function that is to be called. See the complete example with event binding below:

```

[52]: class HelloWorldApp:

    def __init__(self,parent):
        self.parent = parent

```

```

self.container1 = tk.Frame(parent)
self.container1.pack()
self.hwtext = tk.Label(self.container1)
self.hwtext.configure(text="", background="white")
self.hwtext.pack(side="left")

self.container2 = tk.Frame(parent)
self.container2.pack()
self.hwbutton = tk.Button(self.container2)
self.hwbutton["text"] = "Say hello!"
self.hwbutton.pack(side="left")
self.hwbutton.bind("<Button-1>", self.hwbuttonClick)
self.gbbutton = tk.Button(self.container2,
                           text="Goodbye!", background="red")
self.gbbutton.pack(side="left")
self.gbbutton.bind("<Button-1>", self.gbbuttonClick)

def hwbuttonClick(self, event):
    self.hwtext.configure(text="Hello World!")

def gbbuttonClick(self, event):
    self.parent.destroy()

root = tk.Tk()
hwapp = HelloWorldApp(root)
root.mainloop()

```



```

[50]: %%HTML
<style>
td {
    font-size: 20px
}
</style>

```

<IPython.core.display.HTML object>

The full set of event types is rather large, but a lot of them are not commonly used. Here are most of the ones you'll need:

<Button-1>

Button 1 is the leftmost button, button 2 is the middle button (where available), and button 3

the rightmost button. <Button-1>, <ButtonPress-1>, and <1> are all synonyms. For mouse wheel support under Linux, use Button-4 (scroll up) and Button-5 (scroll down)

<B1-Motion>

The mouse is moved, with mouse button 1 being held down (use B2 for the middle button, B3 for the right button).

<ButtonRelease-1>

Button 1 was released. This is probably a better choice in most cases than the Button event, because if the user accidentally presses the button, they can move the mouse off the widget to avoid setting off the event.

<Double-Button-1>

Button 1 was double-clicked. You can use Double or Triple as prefixes.

<Enter>

The mouse pointer entered the widget (this event doesn't mean that the user pressed the Enter key!).

<Leave>

The mouse pointer left the widget.

<FocusIn>

Keyboard focus was moved to this widget, or to a child of this widget.

<FocusOut>

Keyboard focus was moved from this widget to another widget.

<Return>

The user pressed the Enter key. For an ordinary 102-key PC-style keyboard, the special keys are Cancel (the Break key), BackSpace, Tab, Return (the Enter key), Shift\_L (any Shift key), Control\_L (any Control key), Alt\_L (any Alt key), Pause, Caps\_Lock, Escape, Prior (Page Up), Next (Page Down), End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num\_Lock, and Scroll\_Lock.

<Key>

The user pressed any key. The key is provided in the char member of the event object passed to the callback (this is an empty string for special keys).

a

The user typed an "a". Most printable characters can be used as is. The exceptions are space (<space>) and less than (<less>). Note that 1 is a keyboard binding, while <1> is a button binding.

<Shift-Up>

The user pressed the Up arrow, while holding the Shift key pressed. You can use prefixes like Alt, Shift, and Control.

#### <Configure>

The widget changed size (or location, on some platforms). The new size is provided in the width and height attributes of the event object passed to the callback.

#### <Activate>

A widget is changing from being inactive to being active. This refers to changes in the state option of a widget such as a button changing from inactive (grayed out) to active.

#### <Deactivate>

A widget is changing from being active to being inactive. This refers to changes in the state option of a widget such as a radiobutton changing from active to inactive (grayed out).

#### <Destroy>

A widget is being destroyed.

#### <Expose>

This event occurs whenever at least some part of your application or widget becomes visible after having been covered up by another window.

#### <KeyRelease>

The user let up on a key.

#### <Map>

A widget is being mapped, that is, made visible in the application. This will happen, for example, when you call the widget's `.grid()` method.

#### <Motion>

The user moved the mouse pointer entirely within a widget.

#### <MouseWheel>

The user moved the mouse wheel up or down. At present, this binding works on Windows and MacOS, but not under Linux.

#### <Unmap>

A widget is being unmapped and is no longer visible.

#### <Visibility>

Happens when at least some part of the application window becomes visible on the screen.

Next to the Frame container, the Canvas container mentioned earlier is often useful to use. Basically, it allows for including all kinds of graphics – self-drawn, generated or imported. Here a small example added to our HelloWorldApp:

```
[37]: class HelloWorldApp:

    def __init__(self, parent):
        self.parent = parent
```



```

self.container0 = tk.Canvas(parent, width=100, height=100)
self.container0.create_oval(0,0,100,100,fill="yellow")
self.container0.create_oval(45,45,55,55,fill="red")
self.container0.create_oval(25,25,35,35,fill="blue")
self.container0.create_oval(65,25,75,35,fill="blue")
self.container0.create_arc(25,55,75,80,fill="red",
                           style="arc",start=180,extent=180)
self.container0.pack()

self.container1 = tk.Frame(parent)
self.container1.pack()
self.hwtext = tk.Label(self.container1)
self.hwtext.configure(text="",background="white")
self.hwtext.pack(side="left")

self.container2 = tk.Frame(parent)
self.container2.pack()
self.hwbutton = tk.Button(self.container2)
self.hwbutton["text"] = "Say hello!"
self.hwbutton.pack(side="left")
self.hwbutton.bind("<Button-1>", self.hwbuttonClick)
self.gbbbutton = tk.Button(self.container2,
                           text="Goodbye!", background="red")
self.gbbbutton.pack(side="left")
self.gbbbutton.bind("<Button-1>", self.gbbbuttonClick)

def hwbuttonClick(self,event):
    self.hwtext.configure(text="Hello World!")

def gbbbuttonClick(self,event):
    self.parent.destroy()

root = tk.Tk()
hwapp = HelloWorldApp(root)
root.mainloop()

```

The code now adds another container to the GUI frame, namely a Canvas container on top of the two previously defined containers. We create it with `width = 100` and `height = 100`, meaning that the canvas will have a size of 100x100 pixels. Onto that canvas, we draw a big yellow circle, one red and two blue circles as well as an arc, which together create a nice smiley face. :-) Note that the coordinate system of the canvas starts in the upper left corner, so `x,y=0,0` is the coordinate in the upper left, `100,100` the one in the lower left, etc. The drawing functions expect a “bounding box”, i.e. two coordinate pairs that define the rectangle in which the figure is drawn, thus always 4 numbers as parameters in of the methods creating the shapes.



Finally, note that it is also possible to display the name of the app in the window frame, instead of “tk”. All that is required is to add one more line to the main program:

```
root = tk.Tk()
root.title('Hello!')
hwapp = HelloWorldApp(root)
root.mainloop()
```

So much about the basic principles of creating graphical user interfaces in Python with the Tkinter framework. There are a lot more widgets, events and configuration options to be explored, but they follow the same ideas. More advanced information can also be found in the online documentation and tutorials referenced above.

As mentioned in the beginning, several (other) frameworks and toolkits for building GUIs with Python exist. Using toolkits for GUI design often makes it easier to create more “beautiful” interfaces, but on the other hand the code that they generate automatically can be more difficult to understand. If you are interested in more GUI programming, it might nevertheless be worth investigating them further.

## 1.2 Creating Executables with PyInstaller

In the scientific community, people often like to share their Python programs as plain source code, or as Jupyter notebooks, so that they can easily make changes to the program to adapt it to their own data analysis problems. Also, open-sourcing code of computational experiments is in line with reproducibility standards, etc. In other areas, for example commercial software development, the situation is often different. Customers should use the developed software, but not see the code. They should not need a development environment, but just be able to run a stand-alone version of the program (for historical reasons usually called “executable”, although that is not a very intuitive term for interpreted languages like Python, which you can in principle always directly execute).

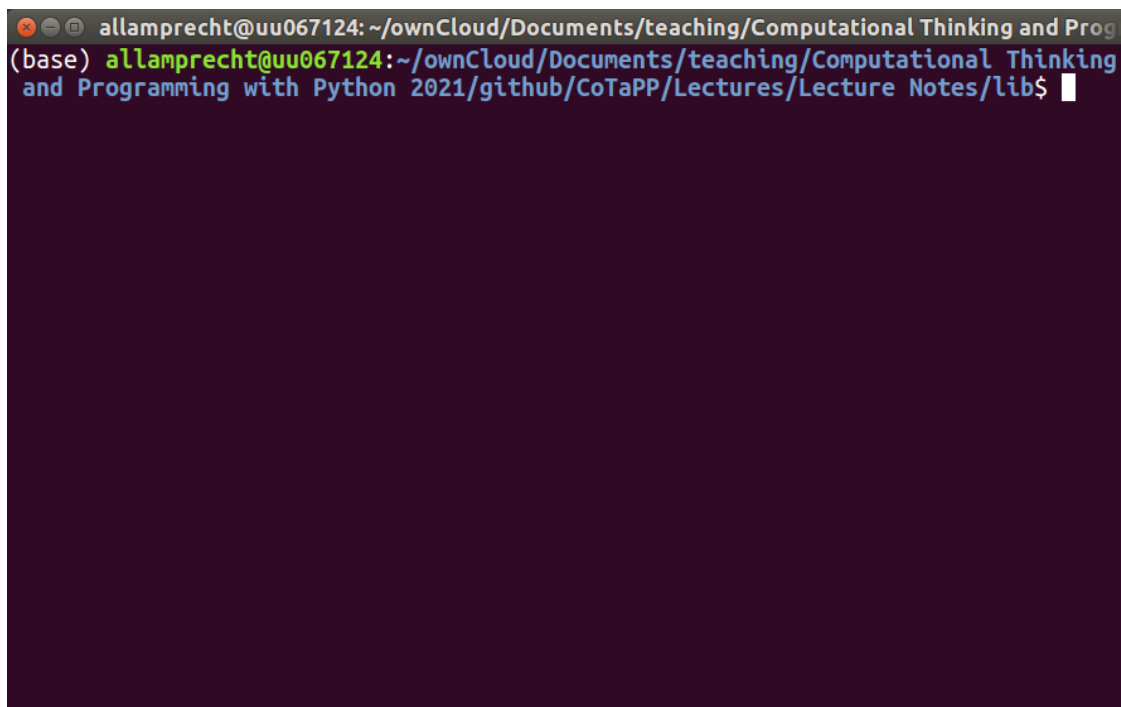
The trick thing with executables (in Python, but also with many other languages) is that they are platform-specific. That is, they can be made for either Windows, Mac OS or Linux platforms, so several versions are needed to make all potential users happy. Furthermore, you can usually only build executables for a specific platform on a machine with the same (kind of) operating system, so professional development teams run different (virtual) machines to be able to do that.

As with many things in the Python ecosystem, there are different frameworks available to “freeze”

(generate executables for) Python programs. PyInstaller (<https://www.pyinstaller.org/>) is one of the few of them that supports all major platforms (e.g. Windows, Mac OS and Linux) and most of the recent Python versions, so it is often a good choice, definitely for a course like this one.

My laptop runs on Linux (Ubuntu 20.04) so we will see how it works there. Generally the process on Windows and Mac OS platforms is the same, but in detail it might differ a bit. The PyInstaller manual at <https://pyinstaller.readthedocs.io/en/stable/> provides quite elaborate instructions for all platforms (on the process, but also regarding requirements and common errors), so please refer to that when you try to build executables for your Python programs.

So, let's assume I have already checked that my system meets the listed requirements. If PyInstaller is not installed already, I can simply do that with the command `pip install pyinstaller` in the terminal. Then it's best to run PyInstaller directly from the directory where the (main) Python program file is located, so I go there first:

A terminal window with a dark background and light-colored text. The prompt shows the user is in a directory path: `allamprecht@uu067124: ~/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib$`. The terminal is currently empty except for the prompt line.

In principle, all I have to do now is to call PyInstaller with the (main) `.py` file of the program that I want to turn into an executable, e.g. `pyinstaller helloworldapp.py`. If the application does not work, try the following line `pyinstaller helloworldapp.py --hidden-import=tkinter -y`. Let's try:

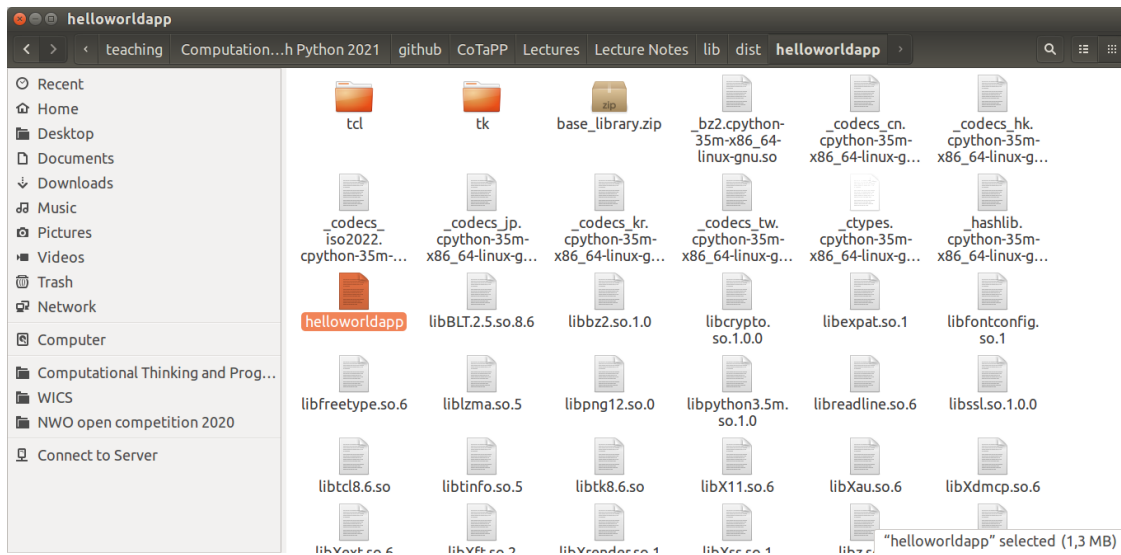
```

allamprecht@uu067124:~/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib$ pyinstaller helloworldapp.py
38 INFO: PyInstaller: 4.2
38 INFO: Python: 3.5.2
38 INFO: Platform: Linux-4.4.0-204-generic-x86_64-with-Ubuntu-16.04-xenial
38 INFO: wrote /home/allamprecht/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib/helloworldapp.spec
41 INFO: UPX is not available.
43 INFO: Extending PYTHONPATH with paths
['/home/allamprecht/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes',
'/home/allamprecht/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib']
53 INFO: checking Analysis
53 INFO: Building Analysis because Analysis-00.toc is non existent
54 INFO: Initializing module dependency graph...
55 INFO: Caching module graph hooks...
58 WARNING: Several hooks defined for module 'win32ctypes.core'. Please take care they do not conflict.
59 INFO: Analyzing base_library.zip ...
3044 INFO: Caching module dependency graph...
3108 INFO: running Analysis Analysis-00.toc

allamprecht@uu067124:~/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib/build/helloworldapp/PYZ-00.pyz
4294 INFO: Building PYZ (ZlibArchive) /home/allamprecht/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib/build/helloworldapp/PYZ-00.pyz completed successfully.
4296 INFO: checking PKG
4296 INFO: Building PKG because PKG-00.toc is non existent
4296 INFO: Building PKG (CArchive) PKG-00.pkg
4314 INFO: Building PKG (CArchive) PKG-00.pkg completed successfully.
4315 INFO: Bootloader /usr/local/lib/python3.5/dist-packages/PyInstaller/bootloader/Linux-64bit/run
4315 INFO: checking EXE
4315 INFO: Building EXE because EXE-00.toc is non existent
4315 INFO: Building EXE from EXE-00.toc
4316 INFO: Appending archive to ELF section in EXE /home/allamprecht/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib/build/helloworldapp/helloworldapp
4320 INFO: Building EXE from EXE-00.toc completed successfully.
4321 INFO: checking COLLECT
4321 INFO: Building COLLECT because COLLECT-00.toc is non existent
4322 INFO: Building COLLECT COLLECT-00.toc
4425 INFO: Building COLLECT COLLECT-00.toc completed successfully.
allamprecht@uu067124:~/ownCloud/Documents/teaching/Computational Thinking and Programming with Python 2021/github/CoTaPP/Lectures/Lecture Notes/lib$

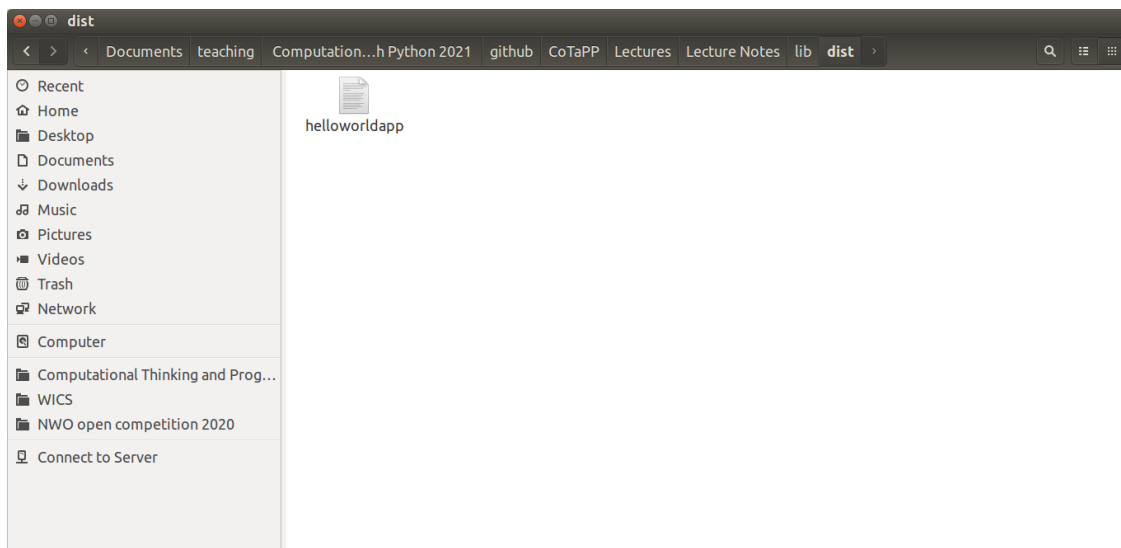
```

A lot of output, but basically it informs us that it has successfully created the executable. It is in the `dist/` directory that it has created in the current directory:



The actual executable file is the one just named **helloworldapp**, executable from the command line by entering `./helloworldapp` (might be slightly different on other platforms, e.g. a **helloworld.exe** file on Windows that can be executed through double-clicking on it). Apparently, there are a lot of other files, too. They are there because the whole required runtime environment (Python interpreter, libraries used, ...) need to be put into the executable, too, so that it is really stand-alone. You don't want your users/customers to deal with a development environment, worry about dependencies, etc.

With the option `--onefile` PyInstaller will pack everything into one single file. That can be easier for users (a single file might look more trustworthy than a large collection of strange-looking files), but in case the program includes related files like a README or License information, they would have to be distributed separately.



Note that Python will automatically detect that modules that are imported by the (main) `.py` file from which the executable is generated also need to be included in the executable. In case further files are to be included, such as README or other files with additional information about the program, or sample input data files, PyInstaller needs to be told about them (e.g. via the

command line). And of course there are many more options and advanced features that can be relevant especially when creating executables for larger, more complex programs, but the PyInstaller website and community should be able to help with that, too.

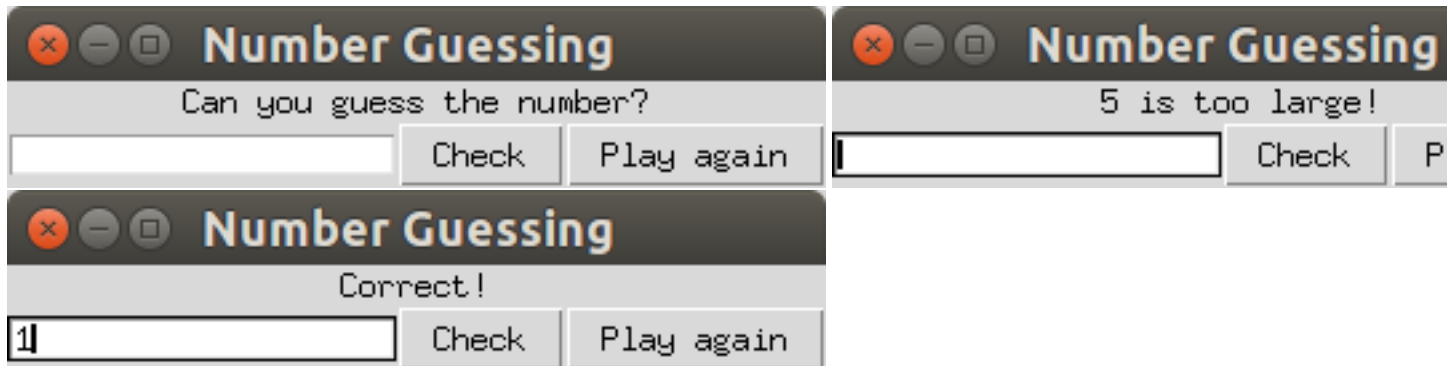
### 1.3 Exercises

Please use Quarterfall to submit and check your answers.

#### 1.3.1 1. Number Guessing with GUI ( )

In an earlier lecture, we programmed a little command-line number-guessing game. The program would generate a random number between 1 and 10, and then ask the user to guess a number until they hit the right one. When the guess is wrong, it would display a message if it is too large or too small.

Now create a simple GUI for playing the number guessing game. It should look something like:

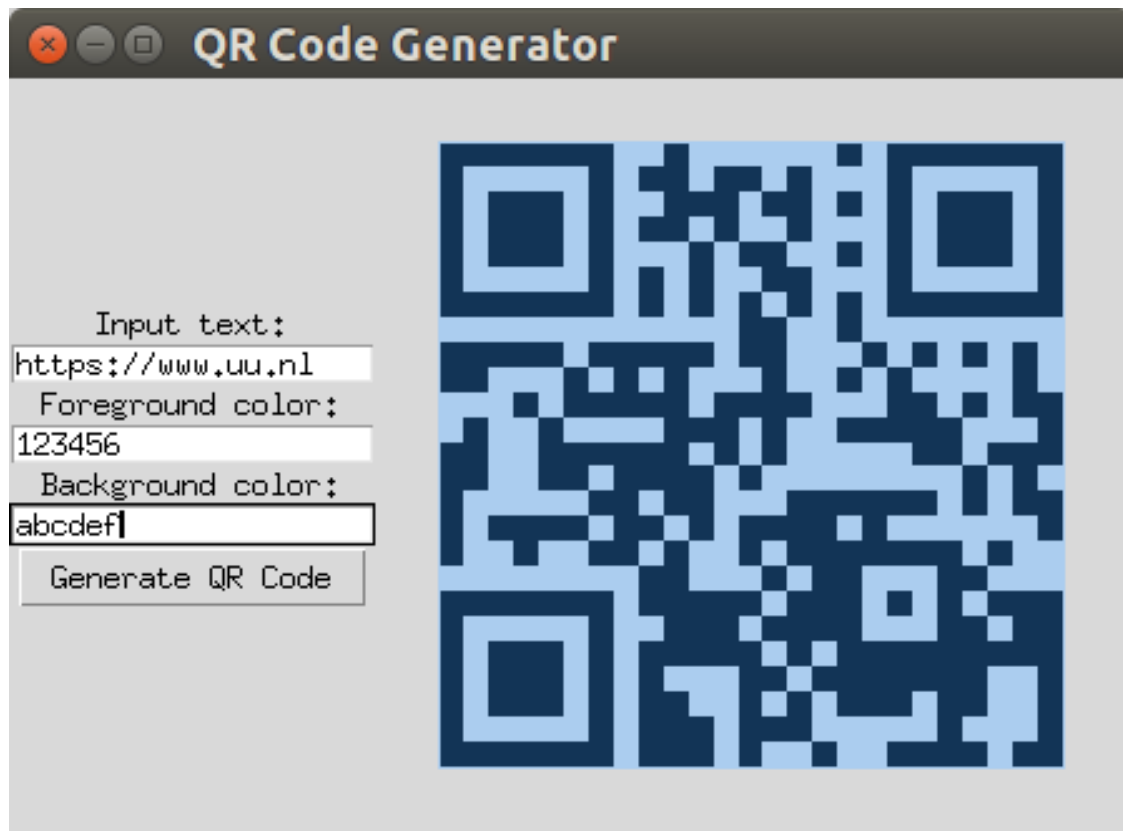


That is, it has a text field (`Label`) to display the different messages. Below this, there are an input field (`Entry`) for the user to enter their guess, a button to check the guess (also triggering the text field to change), and a button to start a new round (generating a new random number and resetting the text and input field).

Optionally, create an executable for this program using `pyinstaller`.

#### 1.3.2 2. QR Code Generator with GUI ( )

In a previous exercise you implemented a command-line client for a QR code generation web service. (If you did not complete the exercise, you can use the code from the sample solution as a basis.) Now use the Tkinter framework to build a graphical user interface (GUI) for the functionality. The user should be able to enter a text and the RGB codes for foreground and background color. After a click on a button the QR code is generated and displayed. (The image does not have to be saved as a file, and for simplicity you can always create and process the image in the same format, for example png). The GUI should look something like (feel free to make a prettier one):



Hint: For displaying the (png) image obtained from the web service on the canvas, the Tkinter.PhotoImage function and the Tkinter.Canvas.create\_image method might be useful.

Optionally, create an executable for this program using pyinstaller.

[ ]: