Learning Objectives: Packaging GUI Applications

Learners will be able to…

- **Create a simple GUI application with Kivy**
- **Compare and contrast bundling command line and Kivy applications**
- **Differentiate creating executables for windowed applications and command line applications**

info

Make Sure You Know

You should be familiar with creating classes in Python, managing projects with Conda, as well as using Pyinstaller for generating executable files.

Limitations

This is not a lesson on building an application with Kivy. This topic will only be covered in very general terms. The goal of this assignment is to demonstrate how to generate executable files for GUI applications built with Python.

GUI Applications

The Kivy Library

You can create graphical user interface (GUI) applications with Python. There are a variety of tools to do this, but we are going to focus on the Kivy library. Kivy allows you to easily create cross-platform applications. Take a look at the gallery to see kinds of projects you can create with Kivy.

Another benefit to using Kivy is that it works with Pyinstaller to create an executable file. To be fair, Pyinstaller also works with other Python GUI libraries like PyQT and Tkinter.

Let's start by creating the `gui` virtual environment with Conda. Enter the following command in the terminal (bottom-left).

```
conda create --name gui -y
```

Once Conda finishes creating the virtual environment, activate it. You should see `(gui)` appear at the beginning of the prompt.

```
conda activate gui
```

Next install the Kivy library. This is not found in the Anaconda repository, so you need to specify the `conda-forge` channel when installing.

```
conda install -c conda-forge kivy -y
```

Application Basics

We are not going to build a sophisticated application with Kivy. In fact, the application is quite boring. However, the idea behind this assignment is to see how Pyinstaller creates GUI executuables. With that in mind, let's build this very simple application that has a counter. Press the button with a `+` to increase the counter by one, and press the `-` button to decrease the counter.

Start by importing the various modules we need. Our Kivy application is going to be a subclass of the `APP` class. We are going to use a simple grid layout, which means our application is divided into columns and rows.

Widgets are placed in the application according to the layout. A `Label` is the widget used for displaying text, and a `Button` widget is used for interaction.

```python
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
```

Create the `ClickButtons` class (which will be the name of our application), which inherits from `App`. Create the `build()` method and define `self.window` as a grid layout with a single column. That means each widget will be stacked on top of one another. The last line of the `build` method should return `self.window`. Finally, create a conditional (outside of the class) that checks if the script is being run directly. If so, call the `run()` method. The comments are there to help us organize our code, they are not required.

```python
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.button import Button

class ClickButtons(App):
  def build(self):
    # define the window
    self.window = GridLayout(cols=1)

    # define the widgets

    # add the widgets

    return self.window

if __name__ == "__main__":
  ClickButtons().run()
```

Believe it or not, that is enough code to generate a GUI. It does not do anything because it is a blank window, but that is all it takes to get started with Kivy. Run the following command in the terminal.

```
python app.py
```

Kivy will return lots of text as it prepares the GUI. If there is a problem with your code, Kivy will return an error message (which are very user friendly when compared to Python) and show you the prompt. If you do not see the prompt (looks something like `~/workspace/app$`), that means your application is running. Click the link below to preview your work.

You should see a new tab appear in the top-left panel. In it, you will see our application. There is a window entitled `ClickButtons`. Our window is hard to see. For starters, it is bigger than the panel and it is a black window on a black background. We will address these issues on the next page. It is important to close the GUI window. If not, Python will keep running the window even though we do not see it. After closing the window, look at the terminal; you should see the prompt. That is how we know our GUI application has stopped running. You can now close the tab that contained the GUI application.

challenge

Try this variation:

- Change the text that appears in the title bar of the Kivy app to `My_First_Kivy_App`.

```
class My_First_Kivy_App(App):
  def build(self):
    # define the window
    self.window = GridLayout(cols=1)

    # define the widgets

    # add the widgets

    return self.window

if __name__ == "__main__":
  My_First_Kivy_App().run()
```

Run the application again and verify that you see the title change in the window. Be sure to change the code back to `ClickButtons` as that will be used in the rest of the code examples.

```
python app.py
```

Deactivate the `gui` virtual environment.

```
conda deactivate
```

▼ **Code**

Your code should look like this:

```
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.button import Button

class ClickButtons(App):
  def build(self):
    # define the window
    self.window = GridLayout(cols=1)

    # define the widgets

    # add the widgets

    return self.window

if __name__ == "__main__":
  ClickButtons().run()
```

Kivy Widgets

Creating Widgets

Activate the `gui` virtual environment.

```
conda activate gui
```

Before we get started with widgets, let’s address the window. It should be smaller and a different color than black. First, import `Window` from `kivy.core.window`.

```
from kivy.core.window import Window
```

Set `Window.clearcolor` to the color we want for the background. We are using hexadecimal (you can also use RGBA) for our colors, so they should be represented as a string. Then set the size of the window using a tuple. Ours will be 150 pixels wide and 300 pixels tall.

```
    # define the window
    Window.clearcolor = '#34495e'
    Window.size = (150, 300)
    self.window = GridLayout(cols=1)
```

Content is added to a window through the use of widgets. There are a variety of widgets available to you. However, we are only going to use some labels and buttons.

A label widget is used to display text on a Kivy application. Create the variable `title` as an instance of a `Label` class. The `text` parameter controls the text that appears in the label. We want our label to say `'Click a Button'`.

```
    # define the widgets
    title = Label(text='Click a Button')
```

Next, create another label, but we want to be able to change the value it displays. Create `self.count` and set its value to zero. Then create the `self.count_label`. The `text` parameter accepts only strings, so typecast `self.count` as a string. Notice that we used `self` in front of the button widget names. We do this because we need to reference this widget outside of the `build` method. We do not interact with or change `title`, so it does not need `self` as a prefix.

```
# define the widgets
title = Label(text='Click a Button')
self.count = 0
self.count_label = Label(text=str(self.count))
```

The last widgets we are going to create are the `add_button` and `subtract_button` widgets. They should be instances of the `Button` class and the text in the button should be a plus sign and minus sign respectively.

```
# define the widgets
title = Label(text='Click a Button')
self.count = 0
self.count_label = Label(text=str(self.count))
self.add_button = Button(text='+')
self.subtract_button = Button(text='-')
```

Adding the Widgets

You will not see the widgets in the Kivy application until you add them to the window. **Important**, the order in which you add the widgets in your code determines the order in which you see them in the application. Since our application has only one column, the widgets will appear one atop another.

Use the `add_widget()` method on `self.window` to add a widget to the GUI. Pass the widget you want to the aforementioned method. We want to first see the title, followed by the add button, the counter, and finally the subtract button.

```
# add widgets
self.window.add_widget(title)
self.window.add_widget(self.add_button)
self.window.add_widget(self.count_label)
self.window.add_widget(self.subtract_button)
```

Run your application and use the link below to see the output. The application is quite boring and plain. If you click the buttons you should see a visual cue that the application registered a click. However, the clicks do not do anything yet.

```
python app.py
```

challenge

Try this variation:

- Change the order of widgets to be in the reverse order.

▼ **Solution**

```
        # add widgets
        self.window.add_widget(self.subtract_button)
        self.window.add_widget(self.count_label)
        self.window.add_widget(self.add_button)
        self.window.add_widget(title)
```

Run your application with the command below and click the link to see the output.

```
python app.py
```

Lastly, deactivate the `gui` virtual environment.

```
conda deactivate
```

▼ **Code**

Your code should look like this:

```python
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.core.window import Window

class ClickButtons(App):
  def build(self):
    # define the window
    Window.clearcolor = '#34495e'
    Window.size = (150, 300)
    self.window = GridLayout(cols=1)

    # define the widgets
    title = Label(text='Click a Button')
    self.count = 0
    self.count_label = Label(text=str(self.count))
    self.add_button = Button(text='+')
    self.subtract_button = Button(text='-')

    # add widgets
    self.window.add_widget(title)
    self.window.add_widget(self.add_button)
    self.window.add_widget(self.count_label)
    self.window.add_widget(self.subtract_button)

    return self.window

if __name__ == "__main__":
  ClickButtons().run()
```

Finishing the Application

Styling the Application

Activate the `gui` virtual environment.

```
conda activate gui
```

Let’s address the bland appearance of the application. First, we want to better position the widgets. They currently fill the window from left to right and top to bottom. We want to leave some padding around the widgets. Set the `size_hint` to 80% (0.8) of the window width and 95% (0.95) of the window height. That means there will be a little more space to the left and right of the widgets than the space above and below them. Then center the widgets both horizontally and vertically so that the empty space is equally distributed around all the widgets. Use a dictionary to place the center of the widgets 50% (0.5) of the width of the window. In addition, place the center of the widgets 50% (0.5) of the height of the window.

```
    # define the window
    Window.clearcolor = '#34495e'
    Window.size = (150, 300)
    self.window = GridLayout(cols=1)
    self.window.size_hint = (0.8, 0.95)
    self.window.pos_hint = {'center_x' : 0.5, 'center_y' : 0.5}
```

Use the following command to run your application. Click the link to see the output. The widgets should have padding around them on all sides.

```
python app.py
```

For the label widgets, we want to increase the font size, make the text bold, and color the text a light gray. **Note**, making the font size for the title 24 is a little too big given the size of the application. You can adjust the size to what you think looks best.

```
    # define the widgets
    title = Label(text='Click a Button',
                  font_size = 18,
                  bold = True,
                  color = '#ecf0f1')
    self.count = 0
    self.count_label = Label(text=str(self.count),
                             font_size = 24,
                             bold = True,
                             color = '#ecf0f1')
```

Next, the buttons will also have an increased font size and bold text. However, the colors will be different. The add button will be green, while the subtract button will be red. Kivy automatically darkens the background colors for buttons, which we do not want. Setting `background_normal` to an empty string does not alter the color.

```
    self.add_button = Button(text='+',
                             font_size = 24,
                             bold = True,
                             background_normal = '',
                             background_color = '#27ae60')
    self.subtract_button = Button(text='-',
                                  font_size = 24,
                                  bold = True,
                                  background_normal = '',
                                  background_color = '#c0392b')
```

Run the program and click the link to see the output. The application should be styled now. However, the buttons do not do anything when clicked.

```
python app.py
```

challenge

Try this variation:

- Adjust the colors of the application. Search for sites that have hexadecimal colors (like this one). Or, if you want colors that work well together, you can use sites (like this) that provide color palettes.

```
python app.py
```

Adding Callbacks

The final thing we need to address is connecting the buttons to the state of the program. We want the number to increase when we click the green button and decrease when we click the red button. To do this, we need to use a callback. This is a method that is attached to a button. More specifically, we can attach a callback to a particular activity like when a button is pressed or is released. Use the `bind()` method to attach a callback to the pressing of each button. The callbacks should go at the end of the `build()` method but before the return statement.

```
      self.add_button.bind(on_press=self.add_callback)
      self.subtract_button.bind(on_press=self.subtract_callback)

      return self.window
```

Outside the `build()` method and still inside the `ClickButtons` class, define the callbacks. Each one should take `self` and `instance` as arguments. For the addition callback, increase `self.count` by one and update the `text` attribute of `self.count_label` with the new value. For the subtraction callback, decrease `self.count` by one and update the label text. Remember, label text must be a string, so typecast `self.count`.

```
    def add_callback(self, instance):
      self.count += 1
      self.count_label.text = str(self.count)

    def subtract_callback(self, instance):
      self.count -= 1
      self.count_label.text = str(self.count)
```

The GUI application should be finished. Execute the command below and click the link to see the output. Click the buttons to make sure the counter properly responds to the clicks.

```
python app.py
```

Lastly, deactivate the `gui` virtual environment.

```
conda deactivate
```

▼ **Code**

Your code should look like this:

```python
from kivy.app import App
from kivy.uix.gridlayout import GridLayout
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.core.window import Window

class ClickButtons(App):
  def build(self):
    # define the window
    Window.clearcolor = '#34495e'
    Window.size = (150, 300)
    self.window = GridLayout(cols=1)
    self.window.size_hint = (0.8, 0.95)
    self.window.pos_hint = {'center_x' : 0.5, 'center_y' : 0.5}

    # define the widgets
    title = Label(text='Click a Button',
                  font_size = 18,
                  bold = True,
                  color = '#ecf0f1')
    self.count = 0
    self.count_label = Label(text=str(self.count),
                             font_size = 24,
                             bold = True,
                             color = '#ecf0f1')
    self.add_button = Button(text='+',
                             font_size = 24,
                             bold = True,
                             background_normal = '',
                             background_color = '#27ae60')
    self.subtract_button = Button(text='-',
                                  font_size = 24,
                                  bold = True,
                                  background_normal = '',
                                  background_color = '#c0392b')

    # add widgets
    self.window.add_widget(title)
    self.window.add_widget(self.add_button)
    self.window.add_widget(self.count_label)
    self.window.add_widget(self.subtract_button)

    self.add_button.bind(on_press=self.add_callback)
    self.subtract_button.bind(on_press=self.subtract_callback)
    return self.window

  def add_callback(self, instance):
    self.count += 1
    self.count_label.text = str(self.count)

  def subtract_callback(self, instance):
    self.count -= 1
    self.count_label.text = str(self.count)

if __name__ == "__main__":
  ClickButtons().run()
```

Bundling an Executable

One-Folder Approach

Activate the `gui` virtual environment.

```
conda activate gui
```

Now that we have a GUI application, it is time to transfrom it into an executable with Pyinstaller. Start by installing Pyinstaller with Conda. Remember, you need to fetch the package from the `conda-forge` channel.

```
conda install pyinstaller -c conda-forge -y
```

We can use Pyinstaller to bundle a Kivy application as an executable. There is a slight change when compared to what we did in the previous assignment. Since our entire Kivy application is contained in a single Python file we do not need to tell Pyinstaller where to look for additional files. We could run `pyinstaller app.py` and create our executable.

However, this means our executable will be named `app` which is not very descriptive. Instead, create a spec file and change the name to `counter`. Since this is a windowed application, use the `-w` flag so your computer will not open a terminal when the executable runs (this assumes you are double-clicking on an icon).

```
pyi-makespec --name counter app.py -w
```

Run Pyinstaller with the newly created spec file to generate the executable.

```
pyinstaller counter.spec
```

By default, Pyinstaller uses the one-folder approach. So the executable will be in the `counter` directory which is in the `dist` directory. Use `./` followed by the path to the executable. Click the link to see the output. Your GUI application should work just as before.

```
./dist/counter/counter
```

One-File Approach

We can also use the one-file approach when bundling an executable. First, let’s delete the `build` and `dist` directories as well as the spec file.

```
rm dist build -r counter.spec
```

Generate the spec file just as before, but add the `--onefile` flag.

```
pyi-makespec --name counter app.py -w --onefile
```

Just as before, generate the executable by running Pyinstaller with the newly created spec file.

```
pyinstaller counter.spec
```

The single executable file `counter` should reside in the `dist` directory. Run it with `./` and the path to the executable. Click the link to see the output.

```
./dist/counter
```

Lastly, deactivate the `gui` virtual environment.

```
conda deactivate
```