Learning Objectives: Pyinstaller

Learners will be able to...

- **Define Pyinstaller**
- **Generate an executable file from a Python script**
- **Differentiate the one-folder and one-file approaches**
- **Create just a spec file**
- **Generate an executable from a spec file**
- **Run an executable file on Linux from the terminal**

info

Make Sure You Know

You are comfortable building your own Python package as well as importing third-party packages and managing virtual environments with `conda`.

Limitations

This material goes over the basics of creating an executable on Linux. You will need to run the same commands on either Windows or a Mac to generate executables for those platforms.

What is Pyinstaller?

Problems with Python Packaging

In previous modules we discussed creating your own Python package and how to manage packages. Left unsaid are the problems when trying to share your work with a wider audience. If you are sharing your work with other Python developers, then there is not really a problem. They understand what it takes to work with third-party packages.

However, try and share the same package with somebody who does not know Python. These simple tasks become all the more cumbersome. Most people are familiar with a graphical user interface and know very little about using the terminal. Traditional methods of creating a distributing Python packages are not user-friendly for non-developers.

Pyinstaller helps us to alleviate these problems. This program bundles a Python package and its dependencies into an executable file, something an average user can more easily run. Best of all, Pyinstaller includes a version of Python so users do not have to worry if their computer does not have Python installed. Pyinstaller can create executable files for Linux, Mac, and Windows. However, you cannot create cross-compiled executables. That is, we are working on a Linux machine so we will create executable files for Linux. If you want your work to run on Windows or Mac, you need to develop on a Windows or Mac machine.

Installing Pyinstaller

Change into the `day` directory. This name might seem a bit odd, but we will use this environment to create a package that prints information about various days.

```
cd day
```

Before installing Pyinstaller, create the virtual environment `day`. We want to keep the development environment for this package separate from our system installation of Python.

```
conda create --name day -y
```

Activate the newly created virtual environment.

```
conda activate day
```

Install the `pyinstaller` package.

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

To verify that this worked, check the version number of Pyinstaller.

```
pyinstaller --version
```

At the time of writing, `conda` installed version `5.1`. Your version may be slightly different.

Finally, deactivate the virtual environment `day`.

```
conda deactivate
```

Building a Package

Building the Today Package

Before we can use Pyinstaller to bundle a package as an executable, we first need a package. We are going to create the `today` package that is a command line interface. It gives you information (day, month, and year) for today, tomorrow, and yesterday In addition, the time will be added for the information regarding today.

Start by activating the `day` virtual environment. Enter the following command in the terminal.

```
conda activate day
```

In the top-left panel, initialize both the `date` and `frame` modules.

```
import today.date, today.frame
```

Next, click the link below to open the `date` module.

Creating the Date Module

The `date` module fetches a Python `datetime` object and then makes any adjustments for the day (today, tomorrow, or yesterday) and timezone. It then extracts the information we want and returns it as a dictionary.

Start by importing `datetime` and `timedelta` from the `datetime` package. `timedelta` helps us adjust the information by day and timezone.

```
from datetime import datetime
from datetime import timedelta
```

Create the function `get_info` that takes the string `day` as the argument. We need to create an offset that is used to represent the day. If `day` is `'yesterday'` then the offset should be `-1`. If `day` is `'tomorrow'`, then the offset should be `1`. Any other value produces an offset of `0` which represents today. Calculate this with a nested ternary operator.

```
def get_info(day):
  offset = -1 if day =='yesterday' else 1 if day =='tomorrow'
        else 0
```

`datetime.now()` returns information about this exact moment in time. However, we need to make some adjustments. First, Python returns time as GMT. You need to adjust the time by the difference from your timezone and GMT. Use this site to calculate the difference. This example uses the Eastern timezone in the US, which is four hours behind GMT. Use `timedelta` and `hours = -4` to represent the Eastern timezone. If your timezone is ahead of GMT, substitute the `-4` with your timezone difference. Then add `offset` to the `datetime` information with `(days = offset)` in another `timedelta`.

```
  dt_obj = datetime.now() + timedelta(hours = -4) +
        timedelta(days = offset)
```

Once our `datetime` information is properly adjusted, extract the day of the week, the numerical date, month, and year. The `strftime()` method returns information as a formatted string. The `%A` modifier returns the full name of the day of the week, `%d` returns the number of the month, `%B` returns the full month, and `%Y` returns the full year.

```
  day = dt_obj.strftime('%A')
  date = dt_obj.strftime('%d')
  month = dt_obj.strftime('%B')
  year = dt_obj.strftime('%Y')
```

Use the `time()` method to extract the time from the `datetime` object. However, this is not a string; Python returns a `datetime` object withthe time. Use the `strftime()` method and the `%X` modifier to return the full time (hour, minute, and seconds) as a string. Finally, return a dictionary that has a key-value pair for the day, date, month, year, and time.

```
  dt_time = dt_obj.time()
  time = dt_time.strftime('%X')
  return {'day' : day, 'date' : date, 'month' : month, 'year' :
        year, 'time' : time}
```

Testing the Script

Let's test our code by creating a conditional that will only run if the `date` module is run directly. We want to check all three variations of our code, so pass `'yesterday'`, `'today'`, and `'tomorrow'` to the `get_info` function.

```
if __name__ == '__main__':
  print(get_info('today'))
```

Run the script by entering the command below into the terminal.

```
python date.py
```

The output of your script is dependent on the day, time, and location of your code. Your output will not match the example below. However, check to make sure the script is returning information that makes sense.

```
{'day': 'Tuesday', 'date': '24', 'month': 'May', 'year': '2022',
'time': '12:09:36'}
```

challenge

Try these variations:

- Test the script with `'yesterday'`. Run the script again with the same terminal command.

```
if __name__ == '__main__':
  print(get_info('yesterday'))
```

- Test the script with `'tomorrow'`. Run the script again with the same terminal command.

```
if __name__ == '__main__':
  print(get_info('tomorrow'))
```

▼ **Code**

Your code should look like this:

```
from datetime import datetime
from datetime import timedelta

def get_info(day):
  offset = 1 if day == 'tomorrow' else -1 if day == 'yesterday'
        else 0
  dt_obj = datetime.now() + timedelta(hours = -4) +
        timedelta(days = offset)
  day = dt_obj.strftime('%A')
  date = dt_obj.strftime('%d')
  month = dt_obj.strftime('%B')
  year = dt_obj.strftime('%Y')
  dt_time = dt_obj.time()
  time = dt_time.strftime('%X')
  return {'day' : day, 'date' : date, 'month' : month, 'year' :
        year, 'time' : time}

if __name__ == '__main__':
  print(get_info('yesterday'))
  print(get_info('today'))
  print(get_info('tomorrow'))
```

Deactivate the `day` virtual environment.

```
conda deactivate
```

Building the Frame Module

Installing the Fire Package

We are going to install the Fire package, which simplifies the creation of a command line interface. Start by activating the `day` virtual environment. Enter the following command in the terminal.

```
conda activate day
```

Then install Fire and its dependencies from the `conda-forge` channel. Now, you are ready to create the `frame` module.

```
conda install fire -c conda-forge -y
```

Frame Module

We are going to format our date and time information so that it appears inside a "frame" with a title. Inside the frame, the individual pieces of information reside on different lines. It should look something like this:

```
-------------------------
|       Today Is:       |
|       ---------       |
|        Tuesday        |
|         24 May        |
|          2022         |
|        12:23:04       |
-------------------------
```

Start by importing the date module as well as the `fire` package. Use the conditional that checks if the script is being run directly and bind the `main` function (we will create this in a bit) to the `Fire` class.

```
import date
import fire

if __name__ == '__main__':
  fire.Fire(main)
```

Declare the `main` function that has the argument `day` with the default value of `today`. The variable `width` represents the size of the "frame". The `line` variable is a series of dashes that run the width of the frame. Call the `get_info()` method from the `date` module and store the dictionary as `info`. Print the "top" of the frame. Call the `display_date` function, which we will create in a bit. Finally, print the "bottom" of the frame.

```
def main(day='today'):
  width = 25
  line = width * '-'
  info = date.get_info(day)
  print(line)
  display_date(day, width, info)
  print(line)
```

The `display_date` function takes the day (yesterday, today, or tomorrow), the width of the frame, and the date and time information. Call the `display_title` function. This function prints the correct title dependent upon the value of `day`. The `center_text` function will center any text we pass it inside the frame. Call this function for the day of the week, the date and month (both on the same line), and the year. Only display the time if `day` is `'today'`.

```
def display_date(day, width, info):
  display_title(day, width)
  center_text(width, info['day'])
  center_text(width, f'{info["date"]} {info["month"]}')
  center_text(width, info['year'])
  if day == 'today':
    center_text(width, info['time'])
```

The `display_title` function takes `day` and `width` as arguments. Call the `center_text` function the appropriate title, verb, and number of dashes (which underline the title).

```
def display_title(day, width):
  if day == 'tomorrow':
    title = 'Tomorrow Will Be:'
  elif day == 'yesterday':
    title = 'Yesterday Was:'
  else:
    title = 'Today Is:'
  line = len(title) * '-'
  center_text(width, title)
  center_text(width, line)
```

Finally, the `center_text` function takes the width of the frame and the text to print. This function is going to print a `|` at the beginning and the end. So subtract 2 from `width` as this is the remaining blank space in which the text will be centered. Use the `center()` string method to center the text. Pass it `width - 2` so it can calculate the number of spaces to place on either side of the text.

```
def center_text(width, txt):
  print(f'|{txt.center(width - 2)}|')
```

Testing the Script

Run the `frame.py` script. Do not pass it any arguments. By default it should return the information for today, including the time.

```
python frame.py
```

challenge

Try these variations:

- Test the script with `'yesterday'`. You should see yesterday's information without the time.

```
python frame.py --day 'yesterday'
```

- Test the script with `'tomorrow'`. You should see tomorrow's information without the time.

```
python frame.py --day 'tomorrow'
```

▼ **Code**

Your code should look like this:

```
import date
import fire

def center_text(width, txt):
  print(f'|{txt.center(width - 2)}|')

def display_title(day, width):
  if day == 'tomorrow':
    title = 'Tomorrow Will Be:'
  elif day == 'yesterday':
    title = 'Yesterday Was:'
  else:
    title = 'Today Is:'
  line = len(title) * '-'
  center_text(width, title)
  center_text(width, line)

def display_date(day, width, info):
  display_title(day, width)
  center_text(width, info['day'])
  center_text(width, f'{info["date"]} {info["month"]}')
  center_text(width, info['year'])
  if day == 'today':
    center_text(width, info['time'])

def main(day='today'):
  width = 25
  line = width * '-'
  info = date.get_info(day)
  print(line)
  display_date(day, width, info)
  print(line)

if __name__ == '__main__':
  fire.Fire(main)
```

Deactivate the `day` virtual environment.

```
conda deactivate
```

One-Folder Approach

Bundling the Executable

Start the `day` virtual environment.

```
conda activate day
```

Now that we have a working Python program, it is time to bundle it up as an executable. We are going to use the default method, which generates an executable and a folder of information. The `frame.py` script is the entry point for our program, so we will run Pyinstaller on this file. By default, this will create an executable called `frame`. That is not a good description of what this program does. We should change the name of the final program. In addition, we need to specify where Pyinstaller can find the `date.py` script so we can import the `get_info` function.

In the terminal, run the following command. Pyinstaller will bundle `frame.py` into an executable. The `--name` flag allows us to rename the program to `day`. Finally, the `--paths` flag tells Pyinstaller where to look for the `date.py` file. Give the full path to the file.

```
pyinstaller frame.py --name day --
        paths='/home/codio/workspace/day'
```

Pyinstaller will prints lots of text to the terminal. We are looking for three pieces of information to let us know the process finished. List all of the contents of the `day` directory. In addition to our Python files, we should see a `day.spec` file as well as two directories called `build` and `dist`.

```
ls
```

The `build` directory contains the files created during the build process. This information can help debug the program should Pyinstaller now work as expected. The `dist` directory contains another directory called `day`. This directory contains bundled program and all of the required files. List its contents. You should see all kinds of files, including the `day` executable (colored green).

```
ls dist/day
```

In Linux, use `./` before the name of an executable program. This is similar to double-clicking on an icon on a Mac or Windows machine. Since the `day` executable is in the `day` directory which is in the `dist` directory, the command would be:

```
./dist/day/day
```

challenge

Try these variations:

- Run the program so that it outputs information about yesterday.

```
./dist/day/day yesterday
```

- Run the program so that it outputs information about tomorrow.

```
./dist/day/day tomorrow
```

Exploring the Pyinstaller Output

People wanting to run your program need the executable file *and* all of the information in the folder. In our case, that means the `day` directory found inside the `dist` directory. You cannot just share the executable file `day` all by itself. It will not run. Users need the entire directory. If you want to share your work, you could zip up the folder to share with others.

We tend to think of an executable file as being a solitary file that can exist on its own. By default, Pyinstaller does not do this. Instead it creates a single folder with all that you need to run the program. While the one-folder approach makes sharing a bit cumbersome, it does allow you greater flexibility when debugging your work.

If everything works as intended, you will not need to interact with the `build` directory. However, there is one file in there that may be of use. In our example, it is called `warn-day.txt`. This text file contains any warnings from the build and bundle process. A common problem with Pyinstaller are modules that it cannot find. If you try to run your executable and see an import error, consult the warn file for more information.

Lastly, the spec file is an important file and deserves a bit of an explanation. The spec file tells Pyinstaller how to build and bundle your scripts. Running Pyinstaller creates a generic spec file. However, in our case, a generic file would not work. We needed to make a few changes, which are done through the `--name` and `--path` flags. Enter the following command to list the contents of the `day.spec` file.

```
cat day.spec
```

In the `Analysis` section, you will see the path we added to help Pyinstaller find the `date.py` file.

```
    pathex=['/home/codio/workspace/today'],
```

In the `EXE` section, you will see the new name to be used with the executable file.

```
    name='day',
```

The spec file is a key component to creating an executable bundle with Pyinstaller. You can add data files, other binary files, set options for specific platforms, etc. See the documentation for more information.

```
conda deactivate
```

One-File Approach

Creating the Spec File

Start the `day` virtual environment.

```
conda activate day
```

While the single-folder approach has advantages from a debugging point of view, it is not very user friendly. Let's create a single executable file with Pyinstaller. But, before we do, let's remove the work done on the previous page. Enter the command below to remove the `build` and `dist` directories.

```
rm dist build -r
```

▼ **Did you notice?**
The remove command uses the `-r` flag. You cannot use `rm` to remove a directory. You first need to remove all of its contents first. The `-r` flag means to remove recursively. So the terminal will go into a directory and remove any files. If there are subdirectories, the terminal will go into that directory and remove any contents. Use the `-r` flag if you want to remove a directory from the command line.

Next, we want to rename the spec file so we can compare this file when using the one-folder approach and the one-file approach. Use the move command to overwrite `day.spec` with `old-day.spec`.

```
mv day.spec old-day.spec
```

On the previous page, we talked about the importance of the spec file. Let's create the spec file separately and then use it to build the executable program as a single file. Replace `pyinstaller` with `pyi-makespec` to just make a spec file. Use the same flags as before to set the name and path. In addition, use the `--onefile` flag to generate a single executable file.

```
pyi-makespec frame.py --name day --
        paths='/home/codio/workspace/day' --onefile
```

Once this is done, list the contents of the directory. You should see the `day.spec` file, but the `build` and `dist` directories do not yet exist.

```
ls
```

Let's compare the two spec files with the `diff` utility.

```
diff -c old-day.spec day.spec
```

Output with `*** 24,36 ****` means lines 24 through 36 in the `old-day.spec` file (because it was passed first to the `diff` utility), while `--- 24,40 ----` means lines 24 through 40 in the `day.spec` file. Any line with a `-` or `+` symbol indicates changes to make the two files identical. A `-` means remove a line from the first file to make them identical. A `+` means add the line to the first file to make them identical. Spec files are quite different when using the one-file and one-folder approaches.

The `pyi-makespec` command is a useful tool in that it generates just a spec file. This file acts like a blueprint for generating your executable. You can then edit this manually to ensure the resulting executable is exactly what you need and want.

Generating the Executable

One you have a spec file, you can use the `pyinstaller` command to generate the executable. Instead of passing the name of the Python script to Pyinstaller pass the newly created spec file.

```
pyinstaller day.spec
```

If we list the contents of the current directory, we should now see the `build` and `dist` directories.

```
ls
```

If we look inside the `dist` directory, we will see something different. Instead of another directory named `day` that contains the executable, you will see just the executable `day`. **Note**, directories are colored blue while executables are colored green.

```
ls dist
```

Our executable is located in the `dist` directory. Use `./` to run our program and followed by the path to the executable. It should run as expected.

```
./dist/day
```

challenge

Try these variations:

- Run the program so that it outputs information about yesterday.

```
./dist/day yesterday
```

- Run the program so that it outputs information about tomorrow.

```
./dist/day tomorrow
```

Finally, deactivate the `day` virtual environment.

```
conda deactivate
```