Practical Coding for Sustainability

Mary Chester-Kadwell
mec31@cam.ac.uk

Exercise 3: Making Programs from Scripts

Contents

License

Introduction

In this exercise we will explore how to turn a simple script into a software program, *either*:

- A command-line program that can be automated; *or*,
- A GUI (graphical user interface) that is easier to use.

In doing so we will also *rework* the script to:

- Make the script more **general** so it can be applied to different data, or be used by different users for their own purposes;
- Remove repetitive code by applying the **DRY** (Don't Repeat Yourself) principle;
- Give the users some more **help** documentation so they know how to use it.

Please *choose either* command-line (Exercises 3.0–3.4) *or* GUI (Exercises 3.5–3.9) for this part of the course.

Of course, if you have time and interest, you can do both exercises, but that is not necessary. If you are not sure which one to choose, read the introductory sections and decide which better suits your own circumstances.

Command-Line Program

Introduction

The main benefits of command-line programs centre around how they can be **automated** to create powerful **pipelines** for your project.

Photo by Mike Benna on Unsplash

- Command-line programs can be run **repeatedly** over a large volume of data without your intervention;
- Different **parameters** (options) can be applied without having to type them all manually;
- The outputs of one program can be **piped** to another program as an input, creating a chain of programs;
- You can **log** the output of command-line programs and save the logs to record what you did or debug a problem;
- You can even **schedule** command-line programs to be run at particular times.

Once your script has been turned into a command-line program the possibilities are endless!

Handling Command-Line Options

Strictly speaking we already have a command-line program. When you run the script `named_entity_recognition.py` as we have done you are invoking it from the command-line as a program that is run by the Python interpreter:

```
(venv) $ python bin/named_entity_recognition.py
```

At the moment, though, it does not take any additional options, which means that we are stuck with the file paths that are hard coded in the script for the letter and the HTML output file. If we wanted to change those we would need to modify the script itself every time we used it.

Parameters and Flags

Instead, we want to take file paths specified by the user when they call the script, like this:

```
(venv) $ python bin/named_entity_recognition.py -s
data/henslow/letters_152.xml
```

Notice that the letter file path is preceded by a **flag** `-s` that tells the program to expect a particular parameter, in this case, the **s**ource file. Flags often start with the same letter as the full parameter name they refer to.

We will start by designing the parameters and flags that we want. This is a vital step you should practice before diving into the code and potentially getting lost in the detail. By creating a **specification**, you can think logically about the desired functionality at a high level, and later implement it carefully step by step.

Parameter	**Flags**	**Required?**	**Documentation**
source	`-s --src`	Required	Source file where XML file is found.
html_file	`-h --html`	Optional	Destination file for NER visualisation in HTML format.
json_file	`-j --json`	Optional	Destination file for NER tags in JSON format.

Specification for command-line options

Notice there are two alternative types of flags: short ones with a single dash and single letter (e.g. `-s`); and long ones with two dashes and a word of two or more letters (e.g. `--src`).

There are a number of ways to handle command-line options in Python[1]. One option is to use a third-party library to make it easier and quicker to do so. Given the constraints of time in this course, I have chosen the library **Click**, which we can bolt onto our script to take parameters and add helpful feedback for users.

[1] For a comprehensive overview of all the options refer to the RealPython tutorial Python Command Line Arguments. Note that the textbook *Research Software Engineering with Python* uses the Python standard library module `argparse`.

Program Logic Design

Before we move on to using Click, we should spend a moment thinking about the intended behaviour of our program, and how it will relate to the parameters. At the moment, the script prints out the NER labels to the command line (i.e. what it is known as **standard out**) and it writes an HTML file, whether we want it to or not. Also, it does not currently write the JSON to any file at all.

We now want to change this behaviour and rework the script as follows:

- The source file parameter is required;
- The script always prints out the NER labels to standard out;
- If (and only if) a HTML destination file path is provided a HTML file will be written to that file;
- If (and only if) a JSON destination file path is provided a JSON file will be written to that file.

Reworking the Program

Third-Party Library: Click

Click is "a Python package for creating beautiful command line interfaces" that uses decorators. Decorators are an intermediate Python feature that you may not have come across, but be reassured you don't need to understand decorators to follow this tutorial[2].

As the Click documentation Basic Commands - Creating a Command explains:

A function becomes a command-line command by decorating it with **`@click.command()`**. At its simplest, just decorating a function with this decorator will make it into a callable script:

```
import click

@click.command()
def hello():
    click.echo('Hello World!')
```

Exercise 3.0

We will now apply this principle to our script.

1. Install Click in the virtual environment and freeze the dependencies to `requirements.txt`. (Reminder: What is requirements.txt?) If you find that the

[2] For an in-depth explanation of decorators try the RealPython tutorial Primer on Python Decorators.

GitHub link to the spaCy model has disappeared, add it back in (`https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-2.2.5/en_core_web_sm-2.2.5.tar.gz#egg=en_core_web_sm`)[3].

2. Add the import to the top of the script in an appropriate place.
3. Add the `@click.command()` decorator immediately above the function `main()` with no line in between.

Now test the program:

```
(venv) $ python bin/named_entity-recognition.py
```

So far so good. Although not much seems to have changed. Try asking for some documentation with `--help`.

```
(venv) $ python bin/named_entity_recognition.py --help
```

The output should be something like this:

```
Usage: named_entity_recognition_.py [OPTIONS]

Options:
  --help  Show this message and exit.
```

Click automatically creates information about available options as an aid for the user! When we add the parameters from the specification we designed (above) these will auto-magically appear in the `--help` page without any extra coding required.

Logic for Handling Parameters

Now we need to implement the parameters we designed in the specification.

Exercise 3.1

1. We add options (command-line parameters) with the **`@click.option()`** decorator. These go directly *between* `@click.command()` and the start of the function. For each option we want, add a `@click.option()` decorator. Here is the option for `source`:

```
@click.command()
@click.option(
      "-s",
```

[3] This issue is fixed in the next version of spaCy (v3).

```
        "--src",
        "source",
        type=click.Path(),
        help="""Required. Source file where XML file is found.""",
)
def main():
```

Now add an option for `html_file` and another for `json_file`. Remember to refer to the specification table (above).

2. Click makes it easy for us to perform some basic validity checks on the file path the user will pass to us. Add some options to the type:

```
@click.option(
        "-s",
        "--src",
        "source",
        type=click.Path(
            exists=True,
            dir_okay=False,
            resolve_path=True,
        ),
        help="""Required. Source file where XML file is found.""",
)
```

These particular options mean:

- Check the path exists;
- Check the path is not a directory (we only want a file);
- Fully resolve the absolute path before passing it on.

For more detail see the documentation.

Of course, for `html_file` and `json_file` we don't expect the files to exist already, so just add `resolve_path=True` for these options.

Check that the options have been recognised correctly:

```
(venv) $ python bin/named_entity_recognition.py --help
```

Now we need to make sure that the arguments passed on the command line make it into the functions.

Exercise 3.2

1. Add **`**kwargs`** to the `main()` function so that whatever number of keyword arguments passed from the command line will be passed into the function[4].

```
def main(**kwargs):
```

2. Now we will rework the `main()` function. First, we want to capture the arguments that have been passed in, if any. Let's add something like this at the top of the function:

```
source = kwargs.get("source")
html_dest, json_dest = kwargs.get("html_file"),
kwargs.get("json_file")
```

3. Then we want to add a check that the required `source` argument has indeed been given. If not, we want to tell the user they have made a mistake.

```
if source:
    text = extract_text()
    document = ner(text)
    write_json(document)
    write_html_viz(document)

else:
    click.echo(
        click.style(
            f'Source file is required. For help use "--help"',
            fg='red',
        ),
    )
```

   Click provides this nice way to print messages for the user with `click.echo()` combined with `click.style()`.

Test what happens if you miss out the required `source` argument now:

```
(venv) $ python bin/named_entity_recognition.py
```

[4] Click options are always passed as keyword arguments so we don't need to handle positional arguments. For more about `**kwargs` see the RealPython tutorial Python args and kwargs: Demystified.

4. At the moment the JSON and HTML functions are both called every time. We need to add tests so that they are only called if the relevant command-line option is invoked.

```
if json_dest:
    write_json(document)

if html_dest:
    write_html_viz(document)
```

5. We also need to actually pass the arguments into the functions. Currently the script is still working with the hard-coded file path values. Add them in `main()` like this:

```
text = extract_text(source)
document = ner(text)
if json_dest:
    write_json(document, json_dest)
if html_dest:
    write_html_viz(document, html_dest)
```

   Now review the corresponding functions and adjust the signatures to match. Replace the hard-coded file paths with the arguments.

6. Finally, make sure that `write_json()` actually does write the JSON file. Hint: you can copy and adjust the code for writing to file as used in `write_html_viz()`.

Now test that the script works by trying a range of commands, for example:

```
(venv) $ python bin/named_entity_recognition.py -s
data/henslow/letters_152.xml

(venv) $ python bin/named_entity_recognition.py -s
data/henslow/letters_152.xml -h results/letters_152_viz.html

(venv) $ python bin/named_entity_recognition.py -s
data/henslow/letters_152.xml -j results/letters_152_ents.json
```

Keeping it DRY

You may have noticed that we now have duplication of logic in the `write_json()` and `write_html_viz()` functions. Both functions write to file in an identical way. If at a later point we wished to modify that logic we would have to do so twice, in two different places.

Apart from being a waste of time, this could introduce errors. This makes the code more difficult to maintain and affects how sustainable it is.

The **DRY** (Don't Repeat Yourself) principle is one of the most fundamental in programming. When faced with duplicate code, break out a new function to hold the logic, and call that instead.

Exercise 3.3

1. Let's add a 'private' function intended for internal script use only. In Python all functions are public, but by convention you can indicate your intention that a function be considered 'private' by starting the function name with a single underscore (_).

   Feel free to write your own function, but here is one I wrote you can use:

```
def _write_file(filepath, content):
    output_file = Path(filepath).resolve()
    output_file.open("w", encoding="utf-8").write(content)
```

2. Now remove the duplicate logic in `write_json()` and `write_html_viz()` and replace it with a call to the `_write_file()` function. Something like this:

```
_write_file(json_dest, ents_json)
```

Test that the script still works as expected.

Finish the Job

But wait! We are not done yet.

Exercise 3.4

1. Update the existing docstrings and add a docstring to the new function. When you are updating the docstring at the top of the file think carefully about what the program now does in comparison to the original. Could the program be used to do NER on XML files other than Henslow Correspondence Project letters? Test to make sure that the docstrings work properly.
2. Add some text to the `README.md` that describes the purpose of the project. If you are unfamiliar with the text styling markup Markdown you can consult this GitHub guide Mastering Markdown.

3. Finally, commit your changes to git with a suitable message. Think carefully about what you have achieved by reworking the code before composing the commit message. (Reminder: How to write helpful commit messages.) Push all your changes to the repo on GitHub.

Congratulations! You have successfully:

- Created a command-line program with options from a series of Jupyter Notebook cells;
- Practiced a version-controlled coding workflow suitable for everyday use;
- Applied fundamental programming strategies:
    - Creating a specification;
    - Refactoring and reworking code;
    - Generalising code;
    - Keeping it DRY (Don't Repeat Yourself).
- Improved the sustainability of code by documenting it with:
    - Self-documenting function and variable names;
    - Docstrings.

See Appendix 1 for a sample answer. To maximise your learning, try to resist referring to this until you have attempted your own version! There are many ways to write code that does that same thing. If your code looks very different from the sample answer that doesn't necessarily mean it is 'wrong' as long as it passes your tests.

You might already be thinking of ways to improve the code. Make a note of these to discuss at the next session.

Next Steps

We have just scratched the surface of what command-line programs can do, how they can be used, and how best to write them for sustainability and reproducibility. Now that you have the basics you can explore further in your own time.

The textbook *Research Software Engineering with Python* goes into more detail about:

- Automating with make: **make** is a program (a build manager) you use to run other programs. You can do things like process many files, re-do analyses every time new data arrives, and run analyses that have several steps in a particular order to create a pipeline;
- Testing Software: In our exercises we tested everything by hand — by trying out commands on the command-line. However, this is not a sustainable practice as the size of a program grows and it risks missing problems. Testing is a large topic but even a little time spent writing a few tests can save you time later on debugging.

- Handling Errors: If you have written quite a bit of Python code before you may have noticed that our code has very little error handling. Catching exceptions and writing helpful error messages for the user is an important part of making code more usable and sustainable.

For more on how Python can help with your day-to-day workflow I recommend the book (and accompanying video course on Udemy) *Automate the Boring Stuff with Python*.

GUI Program

Introduction

The main benefits of a GUI (Graphical User Interface) are how **usable** and pleasant they can be for users. The 'users' might be yourself, your immediate collaborators or the wider research community, if you choose to release your program for publication or as open source.

Photo by Aleksandra Sapozhnikova on Unsplash

Even users with significant technical experience can benefit from the convenience of a well-designed GUI program. Your program may be powerful, and your program may be well documented, but if it is difficult to use your users will go elsewhere and your efforts may be wasted.

Human-Computer Interaction

We have all suffered the frustration of clunky, poorly designed GUI programs. Academic software seems to suffer particularly from this affliction! User experience (UX) is a developed and skilled profession of its own and larger software companies will employ staff specifically

in this role. In contrast, most academics are focussed on their research specialties rather than UX or human-computer interaction (HCI) design[5].

Nevertheless, as the author of software of any kind you are equally responsible for the experience users have interacting with your code as you are for what the code actually accomplishes. Even a small amount of mindfulness can result in a dramatically improved interface for your users to enjoy (yes, enjoy!).

UX, HCI and allied practices are large subjects in their own right, like many aspects of software engineering, but two ways to create more usable GUIs are:

1. Plan a design in advance of implementation;
2. Test the design with real users and make improvements based on feedback.

We will cover the first (design) here and leave the second (testing) for another time.

GUI Design

Before we move on to using PySimpleGUI, we should spend a moment thinking about the intended behaviour of our program, and how it will relate to the GUI window. At the moment, the script prints out the NER labels to the command line (i.e. what it is known as **standard out**) and it writes an HTML file, whether we want it to or not. Also, it does not currently write the JSON to any file at all.

We now want to change this behaviour as follows:

- The source file is always required;
- The script always prints out the NER labels somewhere the user can review them;
- If (and only if) a JSON destination folder is provided a JSON file will be written to a file there;
- If (and only if) a HTML destination folder is provided a HTML file will be written to a file there.

These logical inputs and outputs of the program should be mirrored in a top-to-bottom design for the user interface:

Input/Output	**GUI Element (Widget)**
Source file required	File browser for user to pick file
Optional JSON destination folder	Folder browser for user to pick folder
Optional HTML destination folder	Folder browser for user to pick folder

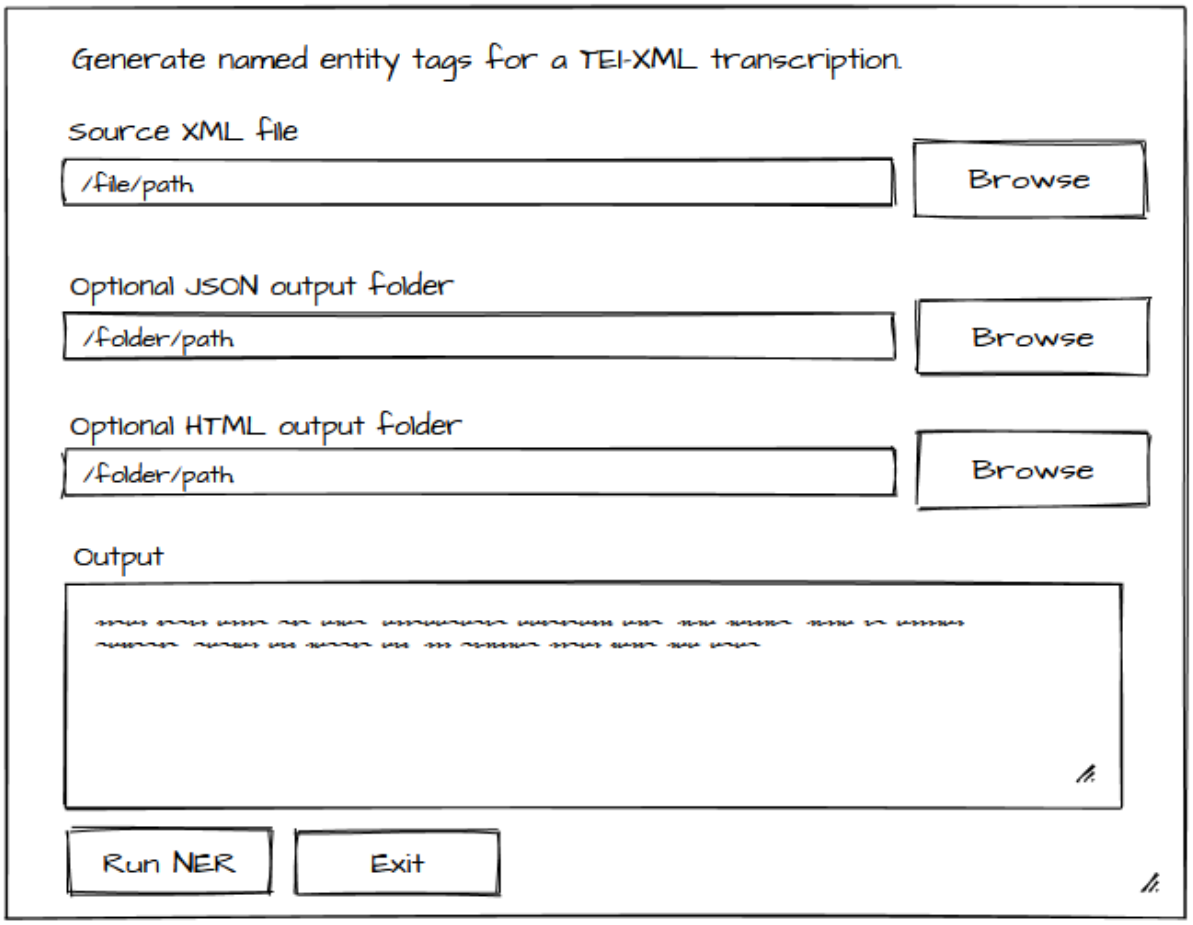
[5] Of course, for some researchers HCI and allied topics are the subject of research itself. We have a group in at the Computer Laboratory in Cambridge that specialises in HCI https://www.cl.cam.ac.uk/research/rainbow/research/

Print out NER labels	Output text area

Additionally, we need:

- Explanatory text — so the user has additional prompts in case the labels are insufficient guidance;
- Buttons — so the user can control the program by running or exiting.

Here is a design sketch that meets these requirements:



We will refer to this during the implementation phase.

Reworking the program

Third-party Library: PySimpleGUI

There are a number of ways to create GUIs in Python[6]. Given the constraints of time in this course, I have chosen the third-party library **PySimpleGUI** to make it easier and quicker to do so. PySimpleGUI takes care of many tricky parts of creating GUIs allowing us to focus on layout, elements and behaviour exclusively.

- **Layout** is implemented as rows and columns in a Python list;
- Many different **elements** (buttons, menus, etc.) are readily available with just one line of code;
- Behaviour occurs in response to **events** (e.g. user clicks a button).

As the PySimpleGUI documentation Jump Start explains, it takes only a few lines of code to create a window.

```
import PySimpleGUI as sg

# Choose a colour theme
sg.theme('DarkAmber')

# Create layout as rows and columns with elements
layout = [  [sg.Text('Some text on Row 1')],
            [sg.Text('Enter something on Row 2'), sg.InputText()],
            [sg.Button('Ok'), sg.Button('Cancel')] ]

# Create the window and pass it the layout
window = sg.Window('Window Title', layout)

# Event loop runs continuously waiting for events to happen
while True:
    event, values = window.read()
    if event == sg.WIN_CLOSED or event == 'Cancel':
        break
        print('You entered ', values[0])

# Close the window when everything is done
window.close()
```

[6] For a range of alternatives refer to the RealPython tutorials PySimpleGUI: The Simple Way to Create a GUI With Python, Qt Designer and Python: Build Your GUI Applications Faster, Python GUI Programming With Tkinter, How to Build a Python GUI Application With wxPython.

Exercise 3.5

This may be a lot to take in at once, but we'll go through it step by step for our script.

1. Install PySimpleGUI in the virtual environment and freeze the dependencies to `requirements.txt`. (Reminder: What is requirements.txt?) If you find that the GitHub link to the spaCy model has disappeared, add it back in (`https://github.com/explosion/spacy-models/releases/download/en_core_web_sm-2.2.5/en_core_web_sm-2.2.5.tar.gz#egg=en_core_web_sm`)[7].

2. Add the import to the top of the script in an appropriate place.

3. Now we will look at the theme demo to pick a colour you like. This will also serve to ensure we have installed PySimpleGUI (and its dependencies) correctly. Open an interactive Python interpreter on the command line:

```
(venv) $ python
```

   Then import PySimpleGui and run the theme previewer to pop up a large window with all available themes:

```
>>> import PySimpleGUI as sg
>>> sg.theme_previewer()
```

   Pick a theme you like and add it to the top of the `main()` function, for example:

```
def main():
    sg.theme("LightGreen")
```

4. Next we'll create the layout of our window. Referring to the GUI design we created (above) we need to add:
   - Some explanatory text;
   - A way of choosing the source XML file;
   - A way of choosing a folder in which to put the JSON and/or HTML file;
   - An output area for the printed NER tags;
   - 'Run' and 'Exit' buttons.

The way PySimpleGui creates a layout is with a **list** of **elements**. There are many elements available, covering everything from text and buttons to checkboxes and images. For example:

```
sg.Text("This text will appear in the window.")
```

[7] This issue is fixed in the next version of spaCy (v3).

Every element that takes a value or generates an event must have a unique **key** so that it can be referred to. For example:

```
sg.Submit("OK", key="-SUBMIT-")
```

The PySimpleGUI convention is to use capitals for the key and wrap it in dashes (-).

If you have time I recommend looking through the PySimpleGUI documentation and trying to create a layout list yourself. Otherwise, you can use the following example code and amend it according to your preference.

```
layout = [

    # Explanatory text
    [sg.Text("Generate named entity tags for a TEI-XML
transcription. Entities will print to the output window.")],

    # Separator
    [sg.HorizontalSeparator(pad=(2, 10))],

    # Source file browser
    [sg.Text("Source XML file (required):")],
    [sg.Input(size=(75, 1), enable_events=True,
key="-SOURCE-"),sg.FileBrowse(target="-SOURCE-", key="-SRC
BROWSE-"),],

    # Separator
    [sg.HorizontalSeparator(pad=(2, 10))],

    # Explanatory text
    [sg.Text("Optionally, you can also output tags as JSON and/or
as a visualisation in HTML.")],

    # JSON folder browser
    [sg.Text("Output JSON folder (optional):")],
    [sg.Input(size=(75, 1), enable_events=True, key="-JSON
DEST-"),sg.FolderBrowse(target="-JSON DEST-", key="-JSON
BROWSE-"),],

    # HTML folder browser
    [sg.Text("Output HTML visualisation folder (optional):")],
    [sg.Input(size=(75, 1), enable_events=True, key="-HTML
DEST-"),sg.FolderBrowse(target="-HTML DEST-", key="-HTML
```

```
BROWSE-"),],

    # Separator
    [sg.HorizontalSeparator(pad=(2, 10))],

    # NER output
    [sg.Text('Output:', key='-OUTPUT TEXT-')],
    [sg.Output(size=(90, 10), key='-OUTPUT-')],

    # Buttons
    [sg.Submit("Run NER", key="-SUBMIT-"),
    sg.CloseButton("Exit"),],
]
```

5. Now we create the window and pass the layout to it:

```
window = sg.Window("Named Entity Recognition", layout)
```

6. Just for the moment, comment out the 4 lines of code in `main()` that do the actual NER.

```
# text = extract_text()
# document = ner(text)
# write_json(document)
# write_html_viz(document)
```

7. In order to respond to button presses, file choices and other events in the GUI we need to add a loop that reads events and values from the window, and waits continuously for actions to occur. We break out of the loop if the user clicks on Close or Cancel.

```
while True:
    event, values = window.read()
    if event in (sg.WIN_CLOSED, "Cancel"):
        break
```

8. Finally, we close the window after the loop has exited.

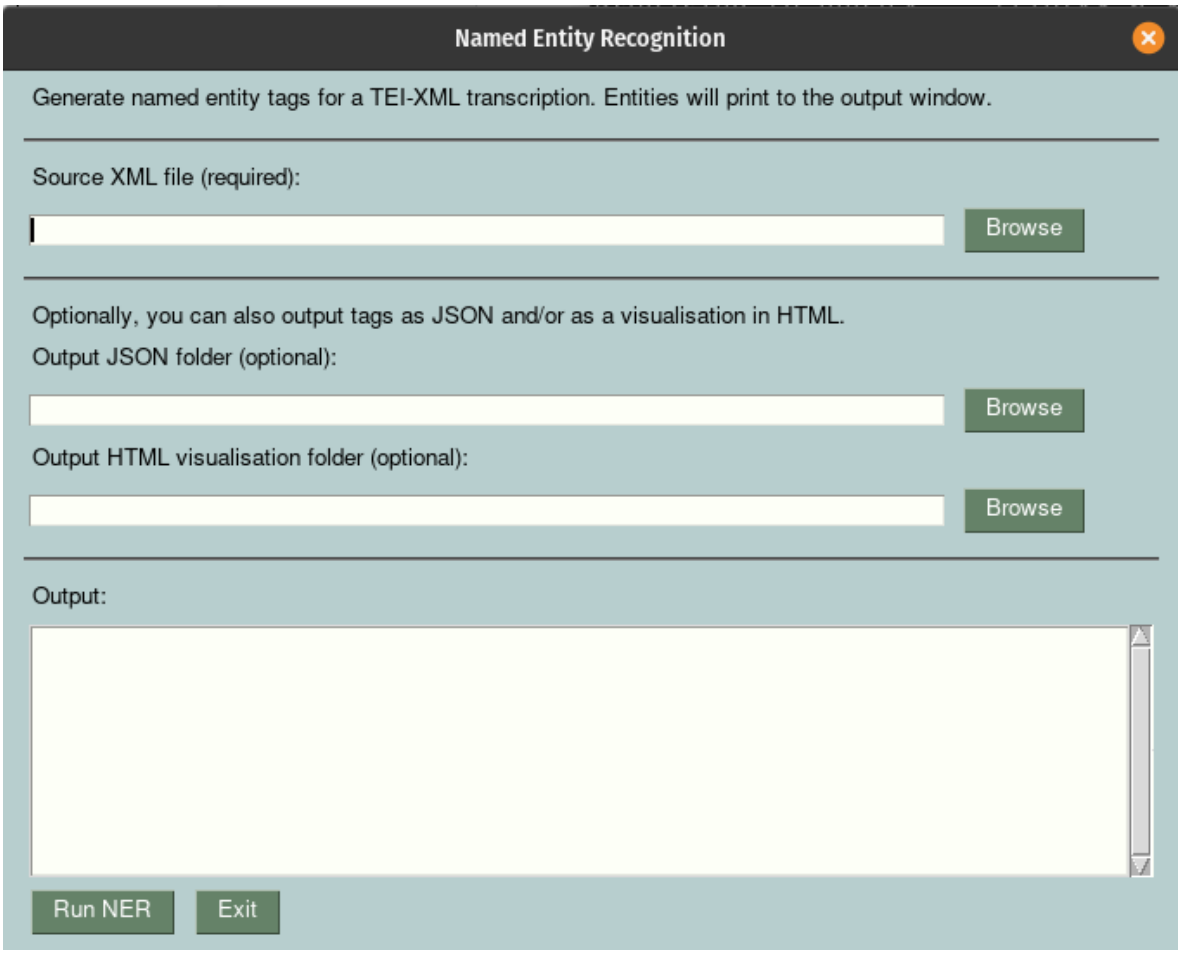
```
window.close()
```

Now test the program to check the window layout:

```
(venv) $ python bin/named_entity_recognition.py
```

As we have commented out the NER code and not included any event behaviour the program currently does nothing! Try the file and folder browsers to test the behaviour. Make any adjustments to the layout that you wish.

You should end up with something a bit like this:



Logic for Handling Events and Values

Now we need to implement responding to events and reading the values from the layout we designed.

Exercise 3.6

1. When the 'Run NER' (`-SUBMIT-`) button is pressed by the user we want to read the values of the XML source file and the output folders, if there are any. Inside the event loop, add in some logic to handle this:

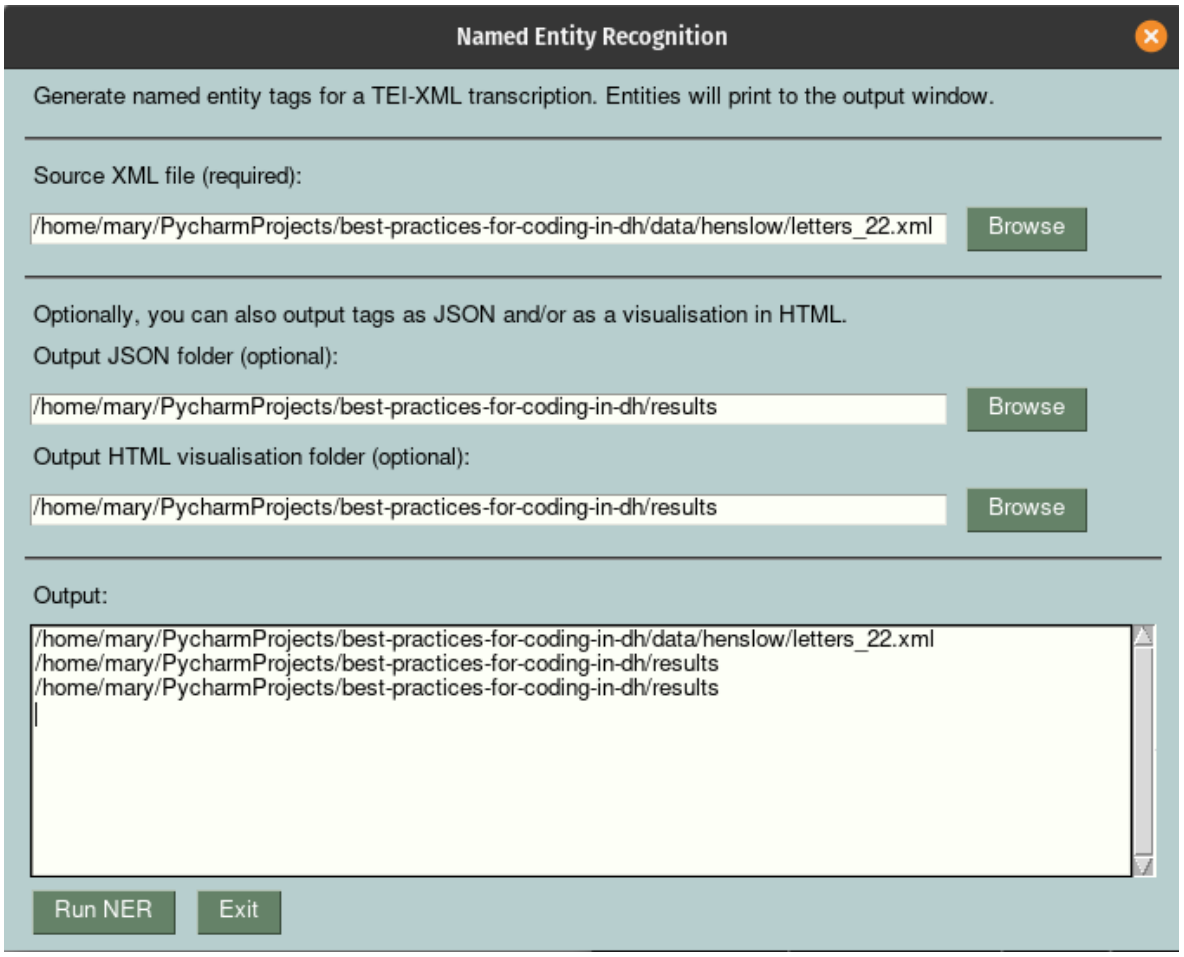
```
if event == "-SUBMIT-":
```

```
source = values["-SOURCE-"]
json_dest = values["-JSON DEST-"]
html_dest = values["-HTML DEST-"]
```

For the purposes of testing it is useful to check that these values are being read correctly, so add in a print line to the `if` block to print the values.

```
print(f'{source}\n{json_dest}\n{html_dest}')
```

Run the program, pick a source file and output folders and click ‘Run NER’ to see the values printed in the ‘Output’ window. It should look something like this:



Remember at this point, the 4 lines of code that do the actual NER should still be commented out.

Now we need to make sure that the file path values from the layout make it into the functions.

Exercise 3.7

1. Uncomment the 4 lines of code that do the actual NER. Now we will rework the `main()` function. First, let's add a check that the required `source` value has indeed been given. If so, the program can continue; if not, we want to tell the user they have made a mistake with a pop-up message.

```
if source:
    json_dest = values["-JSON DEST-"]
    html_dest = values["-HTML DEST-"]

    print(f'{source}\n{json_dest}\n{html_dest}')

    text = extract_text()
    document = ner(text)
    write_json(document)
    write_html_viz(document)

else:
    sg.popup(f'Source XML file is required. Please check and try
again.', title='Oops!', non_blocking=False, keep_on_top=True)
    continue
```

   Run the program again and test what happens if you miss out the required `source` value now.

2. At the moment the JSON and HTML functions are both called every time. We need to add tests so that they are only called if a destination folder is provided.

```
if json_dest:
    write_json(document)

if html_dest:
    write_html_viz(document)
```

3. We also need to actually pass the arguments into the functions. Currently the script is still working with the hard-coded file path values. Add them in `main()` like this:

```
text = extract_text(source)
document = ner(text)
if json_dest:
    write_json(document, json_dest)
if html_dest:
```

```
    write_html_viz(document, html_dest)
```

But wait! We have only provided a folder for the JSON and HTML files, not a file name. We can extract the name (`stem`) of the XML source file and add that to the end of the folder with an appropriate file extension (using `with_suffix()`).[8]

```
source_name = Path(source).stem

if json_dest:
    json_file =
Path(json_dest).joinpath(source_name).with_suffix('.json')
    write_json(document, json_file)

if html_dest:
    html_file =
Path(json_dest).joinpath(source_name).with_suffix('.html')
    write_html_viz(document, html_file)
```

Now review the corresponding functions and adjust the signatures to match. Replace the hard-coded file paths with the arguments.

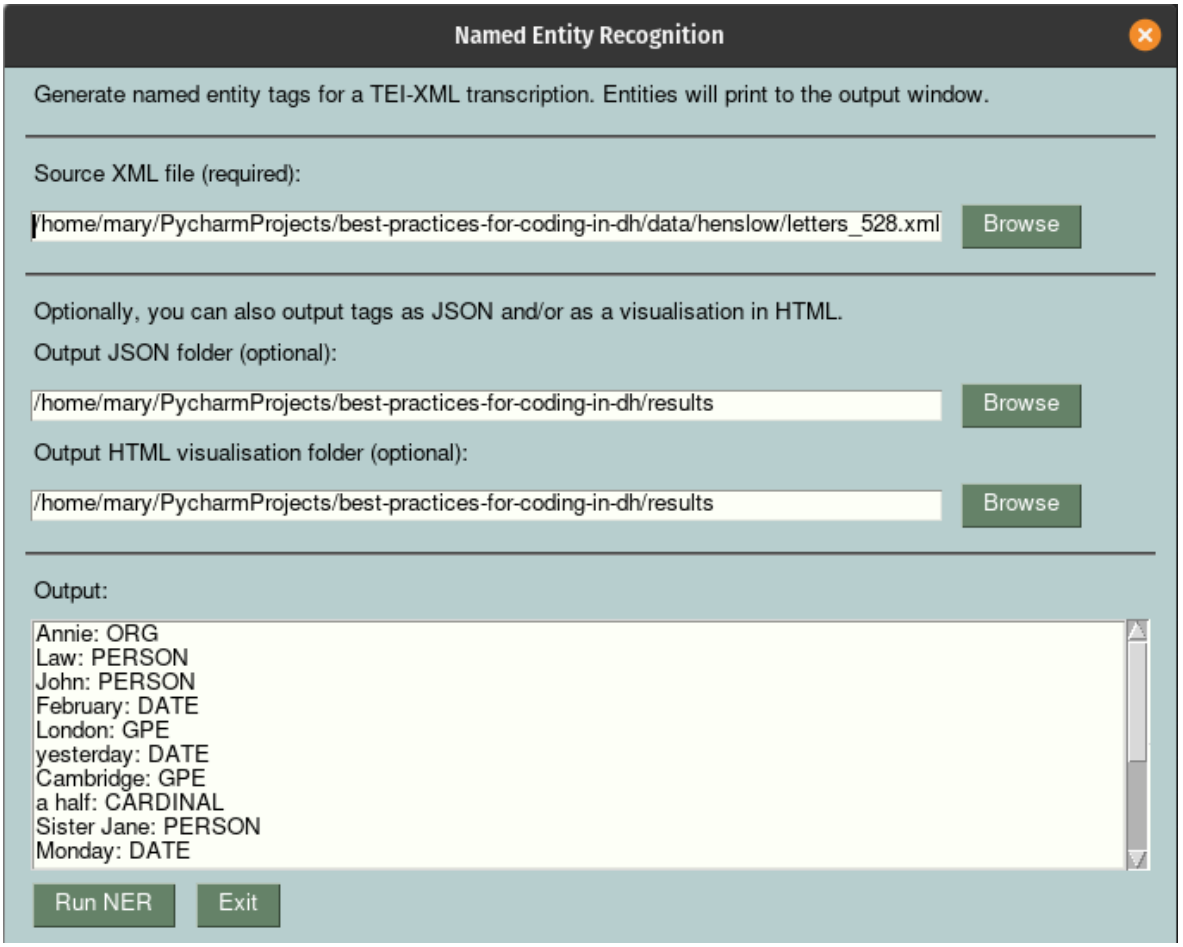
4. Finally, make sure that `write_json()` actually does write the JSON file. Hint: you can copy and adjust the code for writing to file as used in `write_html_viz()`.

Now run the program and test it works by trying a range of test values:

- Just a source XML file;
- Source XML file + HTML folder;
- Source XML file + JSON folder;
- Source XML file + HTML folder + JSON folder.

It should look something like this:

[8] If you are unfamiliar with pathlib from the standard library, I strongly recommend you spend a little time reading up on how to handle file paths as `Path`. It handles the file systems of different operating systems for you and has a huge range of useful methods for handling common file-related tasks.



Check that the JSON and HTML files have appeared and have the contents you expect.

Keeping it DRY

You may have noticed that we now have duplication of logic in the `write_json()` and `write_html_viz()` functions. Both functions write to file in an identical way. If at a later point we wished to modify that logic we would have to do so twice, in two different places. Apart from being a waste of time, this could introduce errors. This makes the code more difficult to maintain and affects how sustainable it is.

The **DRY** (Don't Repeat Yourself) principle is one of the most fundamental in programming. When faced with duplicate code, break out a new function to hold the logic, and call that instead.

Exercise 3.8

1. Let's add a 'private' function intended for internal script use only. In Python all functions are public, but by convention you can indicate your intention that a

function be considered 'private' by starting the function name with a single underscore (_).

Feel free to write your own function, but here is one I wrote you can use:

```
def _write_file(filepath, content):
    output_file = Path(filepath).resolve()
    output_file.open("w", encoding="utf-8").write(content)
```

2. Now remove the duplicate logic in `write_json()` and `write_html_viz()` and replace it with a call to the `_write_file()` function. Something like this:

```
_write_file(json_dest, ents_json)
```

Test that the script still works as expected.

But wait! We are not done yet.

Exercise 3.9

1. Update the existing docstrings and add a docstring to the new function. When you are updating the docstring at the top of the file think carefully about what the program now does in comparison to the original. Could the program be used to do NER on XML files other than Henslow Correspondence Project letters? Test to make sure that the docstrings work properly.
2. Add some text to the `README.md` that describes the purpose of the project. If you are unfamiliar with the text styling markup Markdown you can consult this GitHub guide Mastering Markdown.
3. Finally, commit your changes to git with a suitable message. Think carefully about what you have achieved by reworking the code before composing the commit message. (Reminder: How to write helpful commit messages.) Push all your changes to the repo on GitHub.

Congratulations! You have successfully:

- Created a GUI program (with file path browsers and buttons) from a series of Jupyter Notebook cells;
- Practiced a version-controlled coding workflow suitable for everyday use;
- Applied fundamental programming strategies:
    - Creating a GUI layout;

    - Refactoring and reworking code;
    - Generalising code;
    - Keeping it DRY (Don't Repeat Yourself).
- Improved the sustainability of code by documenting it with:
    - Self-documenting function and variable names;
    - Docstrings.

See Appendix 1 for a sample answer. To maximise your learning, try to resist referring to this until you have attempted your own version! There are many ways to write code that does that same thing. If your code looks very different from the sample answer that doesn't necessarily mean it is 'wrong' as long as it passes your tests.

Next Steps

We have just scratched the surface of what GUI programs can do, how they can be used, and how best to write them for sustainability and reproducibility. Now that you have the basics you can explore further in your own time.

The textbook *Research Software Engineering with Python* goes into more detail about:

- Testing Software: In our exercises we tested everything by hand. However, this is not a sustainable practice as the size of a program grows and risks missing problems. Testing is a large topic but even a little time spent writing a few tests can save you time later on debugging.
- Handling Errors: If you have written quite a bit of Python code before you may have noticed that our code has very little error handling. Catching exceptions and writing helpful error messages for the user is an important part of making code more usable and sustainable.

You may think the GUI design we have is rather clunky! Perhaps you have already spotted areas you would choose to improve. For more on how to design — and test — GUIs I recommend the book *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques* (Access using your Raven account).

Society for Research Software Engineering

If you have enjoyed doing some software engineering, consider joining the Society for Research Software Engineering for further professional development opportunities and community. It's not just for developers; it's for anyone who writes code in a research role. I have found the Slack community and online events incredibly useful and highly recommend it.

Appendix 1: Sample Code

Sample code answer for the command-line program (next page):

```
#!/usr/bin/env python
# coding: utf-8

"""Named Entity Recognition for TEI-XML Transcriptions

This script allows the user to generate named entity tags automatically
for any single English transcription encoded with the TEI-XML tag
<transcription>. Uses the spaCy 'en_core_web_sm' English language model.

Run the program on the command line with the required 'source' option
(-s, --src) and a file path to:

    * Print the named entities to standard out.

Optionally, the program can also write to a specified file:

    * Named entity tags in JSON format (option: -j, --json);
    * Named entity visualisation in HTML (option: -h, --html).

Examples
---------
# Print help text for options
$ python named_entity_recognition.py --help

# Print named entities from source file
$ python named_entity_recognition.py -s data/henslow/letters_152.xml

# Also write named entity tags in JSON format to specified file
$ python named_entity_recognition.py -s data/henslow/letters_152.xml
-j results/letters_152_ents.json

# Also write named entity visualisation in HTML to specified file
$ python named_entity_recognition.py -s data/henslow/letters_152.xml
-h results/letters_152_ents_viz.html.

"""

import json
from pathlib import Path

from bs4 import BeautifulSoup

import click

import spacy
from spacy import displacy
import en_core_web_sm


def _write_file(filepath, content):
    """Write content to specified file.

    Parameters
    ----------
    filepath : str or Path
        The file to write to
    content : str
        The content to write to file

    """

    output_file = Path(filepath).resolve()

    output_file.open("w", encoding="utf-8").write(content)


def extract_text(source):
    """Extract text from an XML file transcription.

    Parameters
    ----------
    filepath : str or Path
        The source XML file

    """

    with open(source, encoding="utf-8") as file:
        letter = BeautifulSoup(file, "lxml-xml")

    transcription = letter.find(type="transcription").text

    return transcription.replace("& ", "and ")
```

```python
def ner(text):
    """Recognise named entities from a text.

    Parameters
    ----------
    text : str
        The text in which to recognise named entities

    Returns
    -------
    document
        a spaCy doc including named entities

    """

    nlp = en_core_web_sm.load()

    document = nlp(text)

    for entity in document.ents:
        print(f"{entity.text}: {entity.label_}")

    return document


def write_json(document, json_dest):
    """Recognise named entities from a text.

    Parameters
    ----------
    document : spaCy doc
        The spaCy doc including named entities
    json_dest : str or Path
        The destination JSON file to write to

    """

    doc_dict = document.to_json()

    ents_dict = {key: value for (key, value) in doc_dict.items() if key == "ents"}

    ents_json = json.dumps(ents_dict)

    _write_file(json_dest, ents_json)


def write_html_viz(document, html_dest):
    """Write a displaCy named entity visualisation to HTML file.

    Parameters
    ----------
    document : spaCy doc
        The spaCy doc including named entities
    html_dest : str or Path
        The destination HTML file to write to

    """

    document.user_data[
        "title"
    ] = "Letter from William Christy, Jr., to John Henslow, 26 February 1831"

    html = displacy.render(document, style="ent", jupyter=False, page=True)

    _write_file(html_dest, html)


@click.command()
@click.option(
    "-s",
    "--src",
    "source",
    type=click.Path(
        exists=True,
        dir_okay=False,
        resolve_path=True,
    ),
    help="""Required. Source file where XML file is found.""",
)
```

```python
@click.option(
    "-h",
    "--html",
    "html_file",
    type=click.Path(
        resolve_path=True,
    ),
    help="Optional. Destination file for NER visualisation in HTML format.",
)
@click.option(
    "-j",
    "--json",
    "json_file",
    type=click.Path(
        resolve_path=True,
    ),
    help="Optional. Destination file for NER tags in JSON format.",
)
def main(**kwargs):

    source = kwargs.get("source")
    html_dest, json_dest = kwargs.get("html_file"), kwargs.get("json_file")

    if source:
        text = extract_text(source)
        document = ner(text)

        if json_dest:
            write_json(document, json_dest)

        if html_dest:
            write_html_viz(document, html_dest)

    else:
        click.echo(
            click.style(
                f'Source file is required. For help use "--help"',
                fg='red',
            ),
        )


if __name__ == "__main__":
    main()

```

Sample code answer for the GUI program (next page):

```python
#!/usr/bin/env python
# coding: utf-8

"""Named Entity Recognition for TEI-XML Transcriptions

This script allows the user to generate named entity tags automatically
for any single English transcription encoded with the TEI-XML tag
<transcription>. Uses the spaCy 'en_core_web_sm' English language model.

Run the program on the command line to pop up the GUI window.

Example
---------
$ python named_entity_recognition.py

In the GUI window, provide a source XML file:

    * Prints the named entities to the output window.

Optionally, provide a destination folder to write named entities to file:

    * Named entity tags in JSON format;
    * Named entity visualisation in HTML.

"""

import json
from pathlib import Path

from bs4 import BeautifulSoup

import PySimpleGUI as sg

import spacy
from spacy import displacy
import en_core_web_sm


def _write_file(filepath, content):
    """Write content to specified file.

    Parameters
    ----------
    filepath : str or Path
        The file to write to
    content : str
        The content to write to file

    """

    output_file = Path(filepath).resolve()

    output_file.open("w", encoding="utf-8").write(content)


def extract_text(source):
    """Extract text from an XML file transcription.

    Parameters
    ----------
    filepath : str or Path
        The source XML file

    """

    input_file = Path(source).resolve()

    with open(input_file, encoding="utf-8") as file:
        letter = BeautifulSoup(file, "lxml-xml")

    transcription = letter.find(type="transcription").text

    return transcription.replace("& ", "and ")

```

```
75
76  def ner(text):
77      """Recognise named entities from a text.
78
79      Parameters
80      ----------
81      text : str
82          The text in which to recognise named entities
83
84      Returns
85      -------
86      document
87          a spaCy doc including named entities
88
89      """
90
91      nlp = en_core_web_sm.load()
92
93      document = nlp(text)
94
95      for entity in document.ents:
96          print(f"{entity.text}: {entity.label_}")
97
98      return document
99
100
101 def write_json(document, json_dest):
102     """Recognise named entities from a text.
103
104     Parameters
105     ----------
106     document : spaCy doc
107         The spaCy doc including named entities
108     json_dest : str or Path
109         The destination JSON file to write to
110
111     """
112
113     doc_dict = document.to_json()
114
115     ents_dict = {key: value for (key, value) in doc_dict.items() if key == "ents"}
116
117     ents_json = json.dumps(ents_dict)
118
119     _write_file(json_dest, ents_json)
120
121
122 def write_html_viz(document, html_dest):
123     """Write a displaCy named entity visualisation to HTML file.
124
125     Parameters
126     ----------
127     document : spaCy doc
128         The spaCy doc including named entities
129     html_dest : str or Path
130         The destination HTML file to write to
131
132     """
133
134     document.user_data[
135         "title"
136     ] = "Letter from William Christy, Jr., to John Henslow, 26 February 1831"
137
138     html = displacy.render(document, style="ent", jupyter=False, page=True)
139
140     output_file.open("w", encoding="utf-8").write(html)
141
142     _write_file(html_dest, html)
143
```

```
144
145 def main():
146
147     sg.theme("LightGreen")
148
149     # Main Layout
150     layout = [
151
152         # Explanatory text
153         [sg.Text("Generate named entity tags for a TEI-XML transcription. Entities will print to the output window.")],
154
155         # Separator
156         [sg.HorizontalSeparator(pad=(2, 10))],
157
158         # Source file browser
159         [sg.Text("Source XML file (required):")],
160         [sg.Input(size=(75, 1), enable_events=True, key="-SOURCE-"),sg.FileBrowse(target="-SOURCE-", key="-SRC BROWSE-"),],
161
162         # Separator
163         [sg.HorizontalSeparator(pad=(2, 10))],
164
165         # Explanatory text
166         [sg.Text("Optionally, you can also output tags as JSON and/or as a visualisation in HTML.")],
167
168         # JSON folder browser
169         [sg.Text("Output JSON folder (optional):")],
170         [sg.Input(size=(75, 1), enable_events=True, key="-JSON DEST-"),sg.FolderBrowse(target="-JSON DEST-", key="-JSON BROWSE-"),],
171
172         # HTML folder browser
173         [sg.Text("Output HTML visualisation folder (optional):")],
174         [sg.Input(size=(75, 1), enable_events=True, key="-HTML DEST-"),sg.FolderBrowse(target="-HTML DEST-", key="-HTML BROWSE-"),],
175
176         # Separator
177         [sg.HorizontalSeparator(pad=(2, 10))],
178
179         # NER output
180         [sg.Text('Output:', key='-OUTPUT TEXT-')],
181         [sg.Output(size=(90, 10), key='-OUTPUT-')],
182
183         # Buttons
184         [sg.Submit("Run NER", key="-SUBMIT-"),
185         sg.CloseButton("Exit"),],
186     ]
187
188     window = sg.Window("Named Entity Recognition", layout)
189
190     # Event Loop
191     while True:
192         event, values = window.read()
193
194         if event in (sg.WIN_CLOSED, "Cancel"):
195             break
196
197         if event == "-SUBMIT-":
198
199             source = values["-SOURCE-"]
200
201             if source:
202
203                 json_dest = values["-JSON DEST-"]
204                 html_dest = values["-HTML DEST-"]
205
206                 text = extract_text(source)
207                 document = ner(text)
208
209                 source_name = Path(source).stem
210
211                 if json_dest:
212
213                     json_file = (Path(json_dest).joinpath(source_name).with_suffix(".json"))
214                     write_json(document, json_file)
215
216                 if html_dest:
217
218                     html_file = (Path(json_dest).joinpath(source_name).with_suffix(".html"))
219                     write_html_viz(document, html_file)
220
```

```
221                 else:
222                     sg.popup(
223                         f"Source XML file is required. Please check and try again.",
224                         title="Oops!",
225                         non_blocking=False,
226                         keep_on_top=True,
227                     )
228                     continue
229
230         window.close()
231
232
233 if __name__ == "__main__":
234     main()
235
```