

Practical Coding for Sustainability

Mary Chester-Kadwell
mec31@cam.ac.uk

Exercise 2: Refactoring and Reworking

Contents

Contents	1
License	3
Introduction	3
Refactoring	3
What is the external behaviour of code?	4
Reworking	5
The Named Entity Recognition Project	7
Introduction	7
Named Entity Recognition (NER)	7
The Jupyter Notebook	9
Inspecting the Dependencies: requirements.txt	9
Exercise 2.0	9
What is requirements.txt?	10
Setting up a Virtual Environment	12
Exercise 2.1	12
Exercise 2.2	14
Running and Understanding the Notebook	18
Exercise 2.3	18
Extracting the Notebook Code	20
Exercise 2.4	20
Creating a Program	22
Writing Functions	22
Code Style and Naming Conventions	22
Function and Variable Names	23
Refactor Script into Small Functions	23
Exercise 2.5	24
Documentation vs Self-Documenting Code	27
Exercise 2.6	28

License

License

This instructional material is copyright © Mary Chester-Kadwell 2021. It is made available under a [Creative Commons Attribution 4.0 International \(CC BY 4.0\) license](#). You are free to re-use this material as long as there is an attribution to the source.

Except where otherwise noted, the example programs and other software are made available under the [MIT license](#). You are free to reuse the code as long as there is an attribution to the source.

Introduction

In this exercise we will introduce some vital skills for any coder: **refactoring** and **reworking** code.



[Photo](#) by Earl Wilcox on Unsplash

Refactoring

Refactoring is taking existing code and improving some aspect of it without changing its essential function. Examples could be:

- Pulling out code that is repeated in many places into one single (private) function (to adopt the 'DRY' principle: Don't Repeat Yourself);
- Rewriting some 'spaghetti' code to make it clearer to understand;

- Modifying an algorithm to speed it up without changing what the algorithm actually does.

The intention is to improve the *internal* quality of the code without changing its *external* behaviour.

What is the external behaviour of code?

When we call a function in code, for example:

```
result = math.sqrt(25)
```

we are using the function `sqrt()` as an external user. We are not concerned with what is going on inside the function's code. We can treat any function as a **black box** where we don't need to look inside it to use it. We only know its inputs and outputs, without any knowledge of its internal workings.



[Photo](#) by Laura Chouette on Unsplash

If we call the function using its name `sqrt` we are guaranteed to be returned the square root of the single number we have provided. This is often known as a **contract**, or in other words, a promise between you and the function: if you provide a valid number, it promises to provide a valid square root.



[Photo](#) by Cytonn Photography on Unsplash

The **signature** of a function is its *name* together with the *number* and *type of arguments* it takes. To use a function, this is all you need to know. This signature is also known as an **Application Programming Interface (API)**.

If you refactored the `sqrt` function, you might change its internal code, but its external behaviour—its API and contract—must remain the same.

Refactoring is usually done to improve the maintainability and quality of the code. You might want to refactor parts of your code prior to releasing them for publication to make it easier to understand and use for others.

Reworking

Strictly speaking, there is a distinction between refactoring and reworking. **Reworking** is a more general term that means editing existing code to change or improve its function and which may change its external behaviour.

Reworking code is something you may do all the time without thinking about it. It may involve fundamental structural change, or it may not. The intention is to evolve your code to achieve some new functionality or behaviour.

Examples could be:

- Reorganising two (public) functions into three functions with new names;
- Merging related functions from several modules into a single module;
- Adding or removing part of the responsibility of an existing function.

In practice, you may not be too concerned with whether your changes are refactoring or reworking code in the strict sense. People often use the two terms interchangeably. But it is

important to be aware of the difference, and to consider refactoring your code regularly as part of your good coding practice.

You could think of reworking code as remodelling your house (perhaps adding new rooms), and refactoring as cleaning and tidying your house (without changing any rooms).

Reworking code



[Photo](#) by Brett Jordan on Unsplash

Refactoring code



[Photo](#) by Austrian National Library on Unsplash

The Named Entity Recognition Project

Introduction

To learn about refactoring and reworking code we will work on a project to start creating an automated **pipeline** for **named entity recognition (NER)**. This is designed to simulate what you might choose to do with your own code projects. While the subject of your research might be quite different from this particular example, the expectation is that the principles and steps are sufficiently general that you should be able to apply them to your own work.

During our NER project we will cover:

- Refactoring and reworking code;
- Dependency management;
- Virtual environments;
- Documentation and self-documenting code;
- Creating a program (either command-line or GUI) from a script (see **Exercise 3**).

Named Entity Recognition (NER)

The purpose of NER is to extract information from unstructured text, especially where it's impractical to have humans read and markup a large number of documents. NER is one natural language processing (NLP) technique that is widely used in digital humanities and data science.

A **named entity** can be any type of real-world object or meaningful concept that is assigned a name or a proper name.



Left to right: [Photo](#) [Photo](#) by simon frederick on Unsplash.

Typically, named entities can include:

- People;
- Organisations;

- Countries;
- Languages;
- Locations;
- Works of art;
- Dates;
- Times;
- Numbers;
- Quantities.

The existing project we will work with uses the Python library [spaCy](#), which is a free and open-source package that can be used to perform automated named entity recognition. In fact, you have already cloned the repository for this existing NER project in **Exercise 1.10**: <https://github.com/mchesterkadwell/named-entity-recognition>

The Jupyter Notebook

The NER project consists of a series of Jupyter Notebooks. Notebooks are extremely popular for data analysis of all sorts, but if you are unfamiliar with Notebooks they can be described as documents that combine executable code, visualisations and explanatory text in one.

For a more in-depth tutorial on getting started with Jupyter Notebooks try this [Jupyter Notebook for Beginners Tutorial](#).

We will run one of the Notebooks in **Exercise 2.3** but first we need to do some setting up.

Inspecting the Dependencies: `requirements.txt`

Exercise 2.0

Go to the `named-entity-recognition` folder (from **Exercise 1.10**) and open the file `requirements.txt` in your text editor of choice.

There you should find a long list of over 90 Python packages that this project depends on to function correctly. These packages are external code, written by someone else, that *may* be called by the `named-entity-recognition` project code in some way.

The start of the file looks something like this:

```
argon2-cffi==20.1.0
async-generator==1.10
attrs==20.3.0
backcall==0.2.0
beautifulsoup4==4.9.3
...
```

I say *may* be called, because many of these packages are dependencies of other packages (i.e. sub-dependencies), and the code in the NER project might not actually call any code that uses these dependencies.

Nevertheless, when you install any Python package, all the packages it relies on are installed as well. Even just installing one or two packages can result in a long list of dependencies for your project.

For the single Notebook we will use (not the whole NER project), here are the primary packages needed:

```
beautifulsoup4==4.9.3
lxml==4.6.2
jupyter==1.0.0
spacy==2.2.4
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
nbconvert==6.0.7
```

Explanation of these packages:

- [Beautiful Soup](#) is a library for getting data out of HTML and XML documents.
- `lxml` is an XML parser used by Beautiful Soup.
- [Jupyter](#) is a package that installs all the other Jupyter packages needed to run Notebooks.
- [spaCy](#) is a Natural Language Processing package.
- The long GitHub URL refers to an English language model used by spaCy.
- [nbconvert](#) is a package for converting Notebooks into other formats.

Notice that each package name is followed by a specific **version number**. These version numbers follow the [SemVer](#) convention.

Specifying an exact version number like this is called **pinning** dependencies. This ensures that when installing the packages for a project exactly the same version of the package is installed as the author intended. There should be no incompatibilities between the packages because the code has been tested and shown to work correctly with these exact versions. When you release code for publication, you should provide pinned requirements¹. This makes the dependencies **reproducible**.

What is `requirements.txt`?

A `requirements.txt` file is a standard file for recording dependencies. It is created and understood by the standard Python **package manager** called `pip`. `Pip` is usually included as standard when you install Python.

¹ Dependency management is a large and complex topic. Be aware that pinning dependencies is not always the right thing to do, unless you are willing to keep every major dependency and their sub-dependencies up-to-date and fully tested on a rolling basis. When developing code on your own or in collaboration you could instead specify your major dependencies in ranges (e.g. `>=2.0, !2.4, <3.0`) and omit the sub-dependencies. This is so that you and your co-developers can update dependencies more easily for security updates and avoid 'dependency hell'. Then when you want to make a reproducible release you can use a lockfile. You might want to take a look at [poetry](#) as an alternative to `pip`.

To install a package with `pip` you use the `install` command:

```
$ pip install beautifulsoup4
```

Note that without giving a version number explicitly, `pip` will install the latest version of the package (in this case, `beautifulsoup4`).

What happens behind the scenes is that `pip` searches [PyPI](#) for you. PyPI is the Python Package Index. You can browse PyPI for packages yourself to find things you might wish to use.



When you have installed all the packages you want you then **freeze** your dependencies into the `requirements.txt` file:

```
$ pip freeze > requirements.txt
```

This creates the list of pinned dependencies for your project.

For a more in-depth tutorial on `pip` see [What is Pip?](#)

If you are using Anaconda, this comes with its own package manager called `conda`. [Conda](#) functions in a very similar way to `pip` with an [install command](#).

```
$ conda install beautifulsoup4
```

To freeze the dependencies to a `requirements.txt` do this:

```
$ conda list -e > requirements.txt
```

It is also possible to install and use `pip` with Anaconda, and we will be doing that in this course for the sake of simplicity. But generally this is *not* recommended on your own

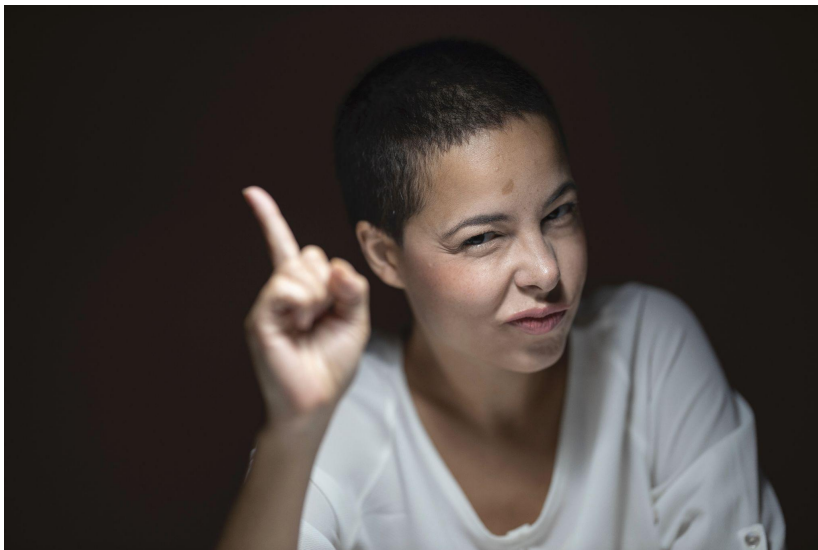
projects if you are using Anaconda exclusively. You should use `conda` with Anaconda and create an [environment file](#) instead.

Setting up a Virtual Environment

Before we can run anything we must install the dependencies we need from the `requirements.txt`. We should do this inside a **virtual environment** to isolate the packages from other projects and avoid interfering with any system-wide packages that are installed.

Conflicting package versions, or the wrong package version, is a major headache that almost everyone will encounter at some point. You may even have spent hours trying to sort out such a mess.

Avoid all that by getting into the good practice of creating a new virtual environment for every single project you start. Yes, every single time you create a new repository, create a new virtual environment and install the dependencies inside it. Do not use global, system-wide packages — ever!



[Photo](#) by engin akyurt on Unsplash

Exercise 2.1

Go to the folder `best-practices-in-coding-for-dh` (you created this in **Exercise 1.1**) and open the file `requirements.txt` in your text editor of choice.

1. Add the following lines of requirements:

```
beautifulsoup4==4.9.3  
lxml==4.6.2
```

```
jupyter==1.0.0
spacy==2.2.4
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
nbconvert==6.0.7
```

Add and commit the change with a suitable message. Refer to [How to write helpful commit messages](#) if you have forgotten!

2. Create a new folder called **data** in the root of your repository. Copy the whole folder **henslow** (and its contents) from the `named-entity-recognition` repository and put it under the `data` directory. Also add an empty **README.md** file in the `data` folder.

By doing this you are adding raw data: a corpus of letters from the nineteenth-century botanist John Henslow. Add and commit the change with a suitable message.²

3. Create a new folder called **results** in the root of the repository.
4. Finally, create a folder called **bin** at the root of your repository. Runnable programs go inside here; 'bin' is an old Unix abbreviation for 'binary', although strictly speaking we won't be using it only for compiled binaries. Copy across the notebook **2-named-entity-recognition-of-henslow-data.ipynb** from the `named-entity-recognition` repository into `bin`.

Once more, add and commit the new file with a suitable message. Inspect your git log.

Your project tree should now look something like this:

```
best-practices-for-coding-in-dh/
├── .gitignore
├── CITATION.md
├── CONDUCT.md
├── CONTRIBUTING.md
├── LICENSE.md
├── README.md
├── requirements.txt
└── bin
```

² For the purposes of this course it is convenient for us to add the data to the repository, but in your project you might not choose to version data with code in your repository for various reasons. Perhaps the data is very large, accessible only remotely, or has a restrictive license in some way. You can add the `data` folder to your `.gitignore`. Versioning data is an active problem of interest in research/data engineering. For example, see [Data Version Control](#).

```

├── 2-named-entity-recognition-of-henslow-data.ipynb
├── data
│   ├── README.md
│   ├── henslow
│   │   ├── README.md
│   │   └── ...
└── results

```

You have now prepared your repository with a runnable script, some raw data to feed into it, and a place to put the results.

Exercise 2.2

We will now create a virtual environment and install into it the packages listed in the `requirements.txt`. The exact instructions depend on whether you are using plain Python or Anaconda.

Plain Python

Open a Terminal or Git Bash for Windows and navigate to the root of the repository.

First check your Python version to make sure you are going to use the correct one. This should be at least version 3.7.

Mac/Linux	Windows
<code>\$ python3 --version</code>	<code>\$ python --version</code>

Create a new virtual environment.

Mac/Linux	Windows
<code>\$ python3 -m venv venv</code>	<code>\$ python -m venv venv</code>

Activate the virtual environment:

Mac/Linux	Windows
<code>\$ source venv/bin/activate</code>	<code>\$ source venv/Scripts/activate</code>

Notice that in Mac/Linux the command prompt will now have **(venv)** in front of it showing that the environment has been activated.

To double check that you are now using the virtual environment you can interrogate it to make sure you are now using the version of Python inside the virtual environment:

```
(venv) $ which python
```

This should return the location of Python inside the `venv` folder.

Then install all the dependencies:

```
(venv) $ pip install -r requirements.txt
```

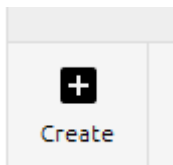
This should initiate a big list of downloads and builds and may take a while to finish. Please be patient.

NB: When you have finished working on your project, you should **deactivate** the environment. *Do not* do this now, but you can refer to the command later:

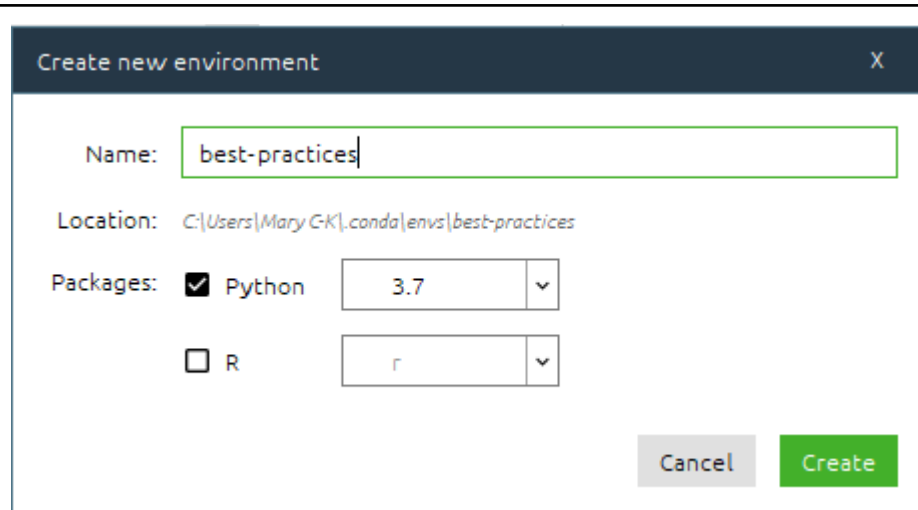
```
(venv) $ deactivate
```

Anaconda

Open [Anaconda Navigator](#). In **Anaconda Navigator** > **Environments** click on the 'Create' button in the bottom left of the Environments list.



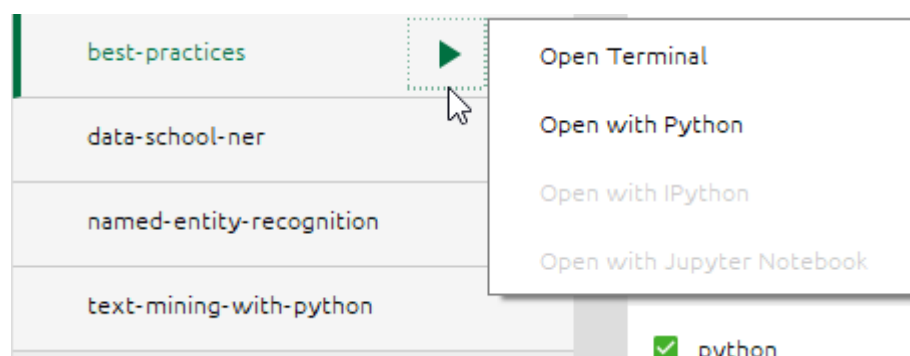
Type a descriptive name e.g. 'best-practices', make sure that 'Python' is checked and under the dropdown pick '3.7'. Make sure that 'R' is left unchecked.



It will take a few seconds to set up...

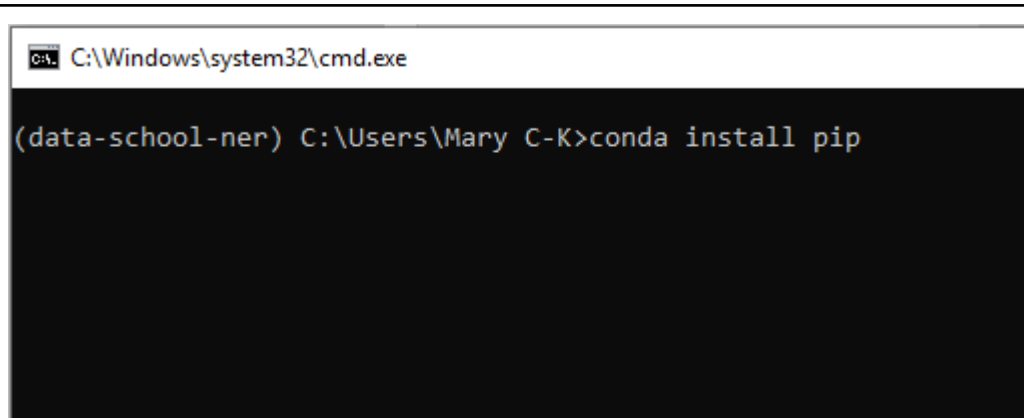
Then in **Anaconda Navigator > Environments** make sure you have selected your new environment.

On the right of the environment name is a small green play arrow. Click on it and pick 'Open Terminal' from the dropdown.



In the Terminal that opens type the following, and press return:

```
> conda install pip
```


A screenshot of a Windows command prompt window. The title bar at the top reads 'C:\Windows\system32\cmd.exe'. The command prompt shows the user is in a directory 'C:\Users\Mary C-K' and has entered the command 'conda install pip'. The output of the command is not visible in the screenshot.

```
C:\Windows\system32\cmd.exe
(data-school-ner) C:\Users\Mary C-K>conda install pip
```

Most likely, `pip` is already installed and you will see this message:

```
# All requested packages already installed.
```

If `pip` is not installed, it will install it.

Then change directory to wherever you saved the `best-practices-in-coding-for-dh` folder.

If you are on **Windows** you will need to use backslashes in the filepath, e.g.
`path\to\named-entity-recognition`






If you are on a **Mac**, make sure to use forward slashes in the filepath, e.g.
`path/to/named-entity-recognition`

Then install all the dependencies by typing:

```
> pip install -r requirements.txt
```

This should initiate a big list of downloads and builds and will take a while to finish. Please be patient.

Now when you go back to the Environments tab in Anaconda Navigator and click the **Update index...** button you should see all the new packages that have been installed into your virtual environment. They come with a little Python symbol next to them to show they have been installed with `pip`.

Installed	Channels	Update index...	Search Packages
Name	T	Description	Version
✓ argon2-cffi			20.1.0
✓ async-generator			1.10
✓ attrs		Attrs is the python package that will bring back the joy of writing classes by relieving you from the drudgery of implementing object protocols (aka dunder methods).	21.2.0
✓ backcall		Specifications for callback functions passed in to an api	0.2.0
✓ beautifulsoup4		Python library designed for screen-scraping	4.9.3

Running and Understanding the Notebook

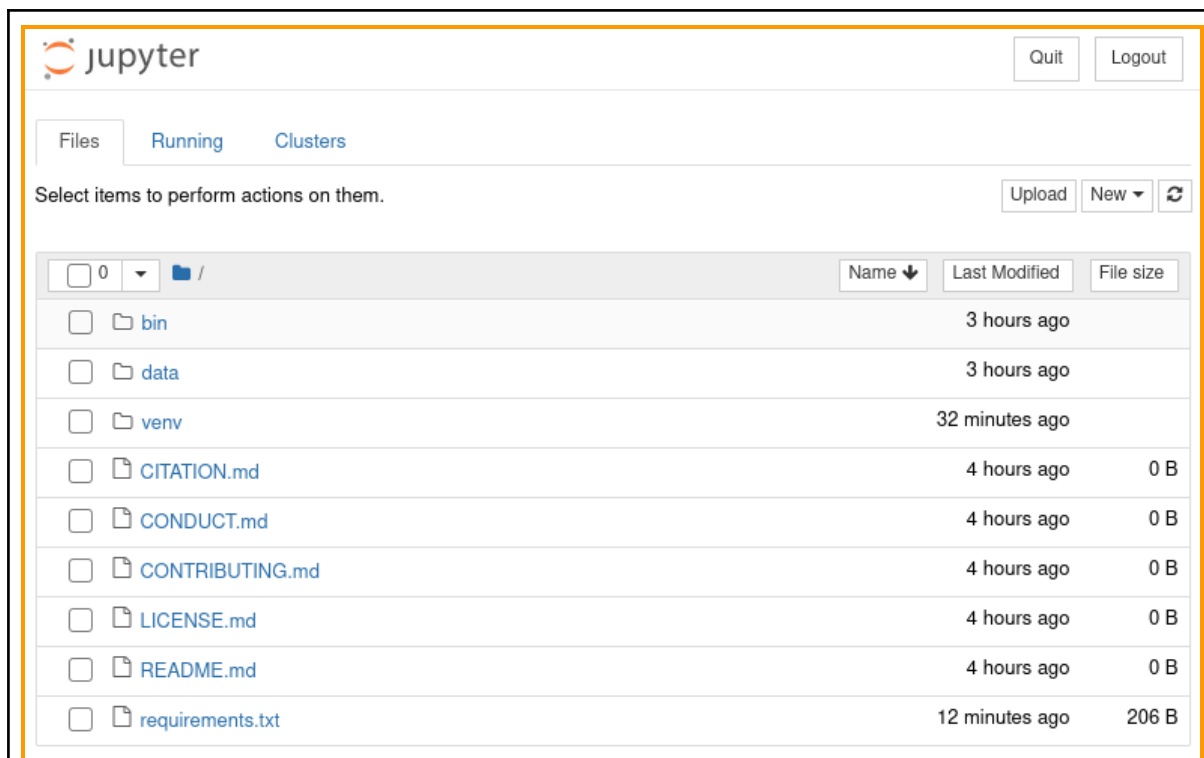
Finally, we can now run the Jupyter Notebook and see what it does!

Exercise 2.3

In the Terminal, Git Bash for Windows or Anaconda Terminal (depending on your platform) make sure you are at the root of your repository and issue the command:

```
(venv) $ jupyter notebook
```

It should automatically open a browser window with the Notebook listing in it, like this:



If not, you can copy and paste one of the URLs in the Terminal window into your browser e.g.

<http://localhost:8888/?token=ddb27d2a1a6cb29a3483c24d6ff9f7263eb9676f02d71075>

(This one will not work on your machine, as the token is unique every time.)

You might normally run your Jupyter Notebooks in a different way. But here we are using a completely isolated Jupyter environment on a version that I have tested works with this Notebook.

In the Notebook listing in your browser, navigate to

`bin/2-named-entity-recognition-of-henslow-data.ipynb` and click on it to open it.

Spend a little time reading through and executing the Notebook cells. (You may need to play with the file paths.) You don't need to completely understand everything, but the key points are that the Notebook:

- Parses an XML letter into a text transcription;
- Runs the text through spaCy automatic NER;
- Outputs the NER tags in JSON format;
- Outputs an HTML visualisation of the NER tags.

Find the line:

```
output_file = Path("output/ent_viz.html")
```

and change it to something like:

```
output_file = Path("results/ent_viz.html")
```

So that the HTML visualisation will be created in our new `results` folder.

When you are eventually finished with the Notebook, press **ctrl+c** to stop the Notebook Server. Then deactivate the environment:

```
(venv) $ deactivate
```

Extracting the Notebook Code

We are going to extract the essential elements of the Notebook code so we can turn into a standalone program.

Exercise 2.4

The library `nbconvert` that we included in the `requirements.txt` can be used to convert Notebooks into a variety of formats, such as static HTML pages and PDFs.

We can convert the Notebook to a script like this:

```
(venv) $ jupyter nbconvert --to script  
bin/2-named-entity-recognition-of-henslow-data.ipynb --output  
named_entity_recognition
```

The output should look like this:

```
[NbConvertApp] Converting notebook  
bin/2-named-entity-recognition-of-henslow-data.ipynb to script  
[NbConvertApp] Writing 23504 bytes to  
bin/named_entity_recognition.py
```

Your project tree should now look something like this:

```
best-practices-for-coding-in-dh/  
├── .gitignore  
├── CITATION.md  
├── CONDUCT.md  
└── CONTRIBUTING.md
```

```
|— LICENSE.md
|— README.md
|— requirements.txt
|— bin
|   |— named_entity_recognition.py
|   |— 2-named-entity-recognition-of-henslow-data.ipynb
|— data
|   |— README.md
|   |— henslow
|       |— README.md
|       |— ...
|— results
```

And the `.py` script we created is a normal executable script you can run on the command line:

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

Two things happen when we run it:

- Prints the entities found in the letter (see line 275 of the script);
- Outputs the HTML visualisation to a file (see line 349 of the script).

(You may find you need to edit the file paths for your system. We will sort this out properly later.)

Creating a Program

In the following exercise (**Exercise 3**) we will turn this script into a program that can:

- Take any single Henslow XML letter as an input;
- Output a JSON file of predicted NER tags to a specified file path;
- Output a HTML file visualisation of predicted NER tags to a specified file path.

You will choose to create either a command-line program that takes command-line arguments, *or* a GUI (graphical user interface) with buttons and a file browser. But before we can do that we need to refactor and rework the code quite a bit into a well-structured ordinary Python script.

Writing Functions

Exercise 2.5 (below) does expect you to have a good grasp of writing functions in Python, although I have given a lot of hints. If you find this too tricky at the moment, you can skip to the example solution at the end. But I do recommend trying the exercise first because the best way to learn coding is to practice!

If you need a reminder of how to write functions in Python try DataCamp [Python Functions Tutorial](#) or the full DataCamp [Writing Functions in Python](#) course.

Code Style and Naming Conventions

Before we start our work, we should review what is considered good, clean **code style** in Python.



[Photo](#) by Jeff Sheldon on Unsplash

The traditional starting point is a document called [PEP8](#), which lays out the coding conventions used in the standard library Python itself. While people do create their own style guides, PEP8 is still very useful, especially for beginners.

The most salient points can be summarised as follows:

- Use 4 spaces per indentation level;
- Imports should be at the top of the file, on separate lines, with similar imports grouped together;
- Function names should be lowercase, with words separated by underscores as necessary to improve readability.

A very helpful tool is the Python package [black](#), which describes itself as the ‘uncompromising Python code formatter’. Running `black` over your code will make everything neat automatically — though of course it won’t improve your function and variable names for you!

If you want to give it a go, first install it in your virtual environment:

```
(venv) $ pip install black
```

Then run it over your code on the command line as often as you like:

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

Function and Variable Names

Naming things is never easy! Good names that accurately and clearly describe what a function does, or what a variable refers to, is part of good coding practice. With good, clear names your code can be **self-documenting**, making it easier for you and others to discern what the code does at a glance.

Here are a few naming tips:

- Make names full words where possible, or clear abbreviations of words where the name would be too verbose;
- Avoid single letters or cryptic acronyms;
- Choose “doing” names for a function: ask yourself, what does the function *do*?
- Choose “being” names for a variable: ask yourself, what *is* the thing it refers to?

Refactor Script into Small Functions

What we are going to do now is *refactor* the script by rearranging its internal order *without* changing anything that it does.

Exercise 2.5

Open the file `named_entity_recognition.py` in your editor of choice.

1. Delete all the comments and code that are not needed. Do add in any comments of your own as you go to make sure you understand what the code does. Don't make any changes to the code at this point. Just strip the script down to core code and group it into lines that have the same general area of responsibility.
2. You should be left with four general areas of responsibility:
 - a. Extract the transcription;
 - b. Process the transcription for NER;
 - c. Dump the NER as JSON;
 - d. Write an HTML file of a NER visualisation.

Make these into four functions with names that describe what they do. Just group the lines of code under functions. At this point don't worry about adding parameters or otherwise changing the code at all.

Don't forget to move all the imports to the top of the file!

3. Now add the following standard code to the bottom of the file:

```
def main():
    pass

if __name__ == "__main__":
    main()
```

What this does is make sure that the `main()` function is only run if the script is called directly (and not run if it is imported). It is a very standard thing to use and you will likely see it when you look at scripts created by others.

`pass` is used to fill empty functions. Empty functions without anything in them at all are not allowed in Python (you will get an error).

4. Next, fill the `main()` function with the steps the script needs to take. It will end up looking something like this (your function names might be different):

```
def main():
    text = extract_text()
    doc = ner(text)
    write_json(doc)
    write_html_viz(doc)
```


5. Notice that the functions `ner()`, `write_json()` and `write_html_viz()` now need to take one argument each. So add one parameter into the function signatures and check that the parameter name matches the name used inside the function.
6. Finally, make sure that the function `extract_text()` returns the transcription text and that the function `ner()` returns the document.
7. Check that the script runs without errors and produces exactly the same output as before:

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

Your script should now look something like this:

```

1  #!/usr/bin/env python
2  # coding: utf-8
3
4  import json
5  from pathlib import Path
6
7  from bs4 import BeautifulSoup
8
9  import spacy
10 from spacy import displacy
11 import en_core_web_sm
12
13
14 # Extract the transcription
15 def extract_text():
16
17     with open("data/henslow/letters_152.xml", encoding="utf-8") as file:
18         letter = BeautifulSoup(file, "lxml-xml")
19
20         transcription = letter.find(type="transcription").text
21
22         return transcription.replace("& ", "and ")
23
24
25 # NER the transcription
26 def ner(text):
27
28     nlp = en_core_web_sm.load()
29
30     document = nlp(text)
31
32     for entity in document.ents:
33         print(f"{entity.text}: {entity.label}")
34
35     return document
36
37
38 # Dump the NER as JSON
39 def write_json(document):
40
41     doc_dict = document.to_json()
42
43     ents_dict = {key: value for (key, value) in doc_dict.items() if key == "ents"}
44
45     json.dumps(ents_dict)
46
47
48 # Write HTML file of the NER visualisation
49 def write_html_viz(document):
50
51     output_file = Path("results/ent_viz.html")
52
53     document.user_data[
54         "title"
55     ] = "Letter from William Christy, Jr., to John Henslow, 26 February 1831"
56
57     html = displacy.render(document, style="ent", jupyter=False, page=True)
58
59     output_file.open("w", encoding="utf-8").write(html)
60
61
62 def main():
63
64     text = extract_text()
65
66     document = ner(text)
67
68     write_json(document)
69
70     write_html_viz(document)
71
72
73 if __name__ == "__main__":
74     main()
75

```

Documentation vs Self-Documenting Code

We have already briefly discussed the concept of **self-documenting code**, that is, where the names of functions and variables help explain what the code does. This is a very good habit to get into from the moment you start writing code, no matter whether you think anyone else will ever see it, or not. Remember: you and your future self both count as users!

In fact, trying to write self-documenting code helps you think more clearly. If you find it is hard to name something, it may be a hint that your function is too big, has too much responsibility, and needs to be split into smaller functions with single responsibilities. This is the principle of [modularity](#).



[Photo](#) by Matt Briney on Unsplash

However, even the best self-documenting code needs a bit of extra help from explicit **documentation**. Documentation means many things:

- Documentation within the code:
 - **Code comments** (lines that start with #);
 - **Docstrings** (special code comments that describe purpose and usage);
- Documentation separate from the main code:
 - **READMEs** and tutorials;
 - Example scripts;
 - Indexes auto-generated from docstrings.

The level of documentation you choose for your project depends on whether it is completely private; private but intended to be public at some point; or actively public. I would argue that you should get into a minimum set of good practices from the start to save yourself a lot of pain and time later on. Any documentation is better than no documentation — as long as it is up to date!

To meet a minimum standard for **sustainability** and **repeatability/reproducibility/reusability**, even in a private project, I would argue code should have:

- A README that describes the purpose of the project;
- Examples of how to use the code;
- Code comments that explain *why* code behaves as it does, if it is not clear;
- Docstrings on all modules, functions and classes.

Docstrings can be minimal on a private project. On a public project, docstrings can be used to generate an index automatically for users to consult. [Sphinx](#) is one automatic documentation generator.

If you have time, I recommend the RealPython tutorial [Documenting Python Code: A Complete Guide](#).

For the moment we will do a minimum set of documentation for our script.

Exercise 2.6

Open the file `named_entity_recognition.py` in your editor of choice, if it is not already open.

Remember that docstrings should use the triple-double quote (`"""`) string format, whether the docstring is multi-lined or not.

1. Write a docstring at the top of the script that:
 - Briefly explains what the script does;
 - Gives an example of how to use the script.

Something like the following:

```

1  #!/usr/bin/env python
2  # coding: utf-8
3
4  """Named Entity Recognition for Henslow Correspondence Project
5
6  This script allows the user to generate named entity tags automatically
7  for any Henslow Correspondence Project XML letter.
8
9  Run the script on the command line with no arguments. It will:
10
11     * Print the named entities to standard out;
12     * Output a named entity visualisation as an HTML file.
13
14  Example
15  -----
16  $ python named_entity_recognition.py
17
18  """
19
20  import json

```

2. Write docstrings for each of the four functions (not `main()`) using [NumPy/SciPy format](#). If the function takes no parameters or returns no value you can omit that section.

You should end up with something like this for each function:

```

41  def ner(text):
42      """Recognise named entities from a text.
43
44      Parameters
45      -----
46      text : str
47          The text in which to recognise named entities
48
49      Returns
50      -----
51      document
52          a spaCy doc including named entities
53
54      """
55

```

Now let's test the docstrings are working as intended by calling `help()` on the functions.

In the Terminal or Git Bash for Windows, change directory into the `bin` folder:

```
$ cd bin
```

Start an interactive Python interpreter session:

Mac/Linux/Windows CMD	Git Bash for Windows
\$ python	\$ python -i

Notice that the command prompt will turn into three angle brackets (>>>):

```
Python 3.8.5 (default, Jan 27 2021, 15:41:15)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Import your module `named_entity_recognition.py`:

```
>>> import named_entity_recognition as ner
```

Call `help()` on any of the functions to see the docstring:

```
>>> help(ner.extract_text)
```

Press **q** to exit the help text.

Once you are finished you can exit the interpreter:

```
>>> exit()
```

Finally, don't forget to commit your changes to git.

Now you can proceed to **Exercise 3**.