Workbook

Witchcraft on Trial

Analyzing Power, Justice, and Legal Narratives through Digital Tools (Python)

Open in GitHub Codespaces

Introduction

This one-week Blended Intensive Program (BIP) seminar brings together students from four partner universities to examine how legal narratives are constructed, contested, and preserved in historical court records. Focusing on the Salem witchcraft trials as a central case study, the seminar combines historical interpretation with hands-on training in digital humanities tools to provide participants with both theoretical insight and practical experience.
Participants will work with digitized court records from the Salem trials to investigate how accusations, testimonies, and verdicts were embedded in the cultural, political, and social tensions of the time. Through this focused case study, the program invites students to consider how power was exercised, resisted, and recorded within the legal system, and how historical sources both reveal and obscure the lived experiences of individuals involved.

A key component of the seminar is the introduction to digital humanities methods that enhance and complicate traditional approaches to historical sources. Digital history tools will be introduced as a means to critically interrogate the sources, uncover hidden patterns, and ask new questions about the structures and dynamics of justice.

The intensive phase will take place at the Kadir Has University in Istanbul with an interdisciplinary and international group of students.Collaborating universities include Bielefeld University, Kadir Has University Istanbul, the University of Vienna and the University of Oslo.

Course Materials

This repository contains the materials and resources for the Digital Training component of the seminar. Various basics will be covered in five sessions, including:

- Introduction to Python programming
- Dealing with GitHub (you're in the middle of it right now, congratulations!) and GitHub Codespaces
- Natural Language Processing (NLP) techniques for text analysis
- Data visualization techniques

- Basic data analysis and interpretation for historical research

Why programming for historians?

Programming is not just for computer scientists – it is a way of thinking that helps historians work with digital sources more efficiently and creatively. It is important to note that programming is not meant to replace traditional historical methods, but rather to complement and enhance them. By learning to program, historians can gain new insights into their sources and develop new ways of asking questions about them.
It allows you to automate repetitive tasks, explore large text collections, and visualize your findings in meaningful ways.

In short:

- **Automation**: Save time by letting your computer handle repetitive work (e.g. counting words, sorting data).
- **Data analysis**: Work with large collections of sources that can't be read manually.
- **Text analysis**: Identify patterns, keywords, and structures in historical texts.
- **Web scraping**: Collect historical materials from online archives and repositories.
- **Visualization**: Present your findings in clear and engaging visual forms.

Why Python?

Python is one of the most accessible programming languages for beginners.
It is widely used in the humanities and data analysis, and its clear, readable syntax makes it easy to learn.
With Python, you can work on a wide range of tasks – from cleaning and analyzing texts to creating visualizations and automating workflows. In other words: Python gives you a whole toolkit for exploring historical data.

Course structure

The aim of the course is to provide you with **basic knowledge of the Python programming language**. The intention is to give you ideas on how you can continue working with digital methods and take away an initial fear of programming. You should take as much in-depth knowledge as possible that can also be followed up at home and, above all, to bring the advantages of programming for humanities tasks closer.
The workshop is structured as follows: There is a folder for each of the following learning units, which contains the content of the session and is designed to be as interactive as possible. This content is then worked on together and deepened with short inputs. As you are currently in the middle of the first learning unit, let's get started right away!

Session 1: Introduction to Python programming

- Basics of Python syntax and structure
- Variables, data types, and operators
- Lists, tuples, and dictionaries

Session 2: Working with Python Libraries and functions

- Control structures (if statements, loops)
- Working with files
- Functions and modular programming
- Libraries and packages

Session 3: Techniques for text analysis

- Introduction to NLP and its applications in the humanities
- Text preprocessing (tokenization, stemming, lemmatization)
- Basic text analysis techniques (word frequency, n-grams, TF-IDF)
- Named Entity Recognition (NER)
- Working with spaCy, NLTK, and Gensim

Session 4: Data visualization techniques

- Importance of data visualization in historical research
- Introduction to Matplotlib
- Creating basic plots
- displaCy for visualizing NLP results

Session 5: Practice with your own projects

- Applying learned techniques to your own historical data
- Troubleshooting and problem-solving

Good to know:

1. GitHub

GitHub is a platform that has become an integral part of modern collaborative software development, but is now also used by many other teams as a version control tool. In a nutshell, GitHub is a web-based platform that offers version control and collaboration in the development of software projects. GitHub acts as a centralized hub for managing and collaborating on code. It utilizes a version control system called Git, which tracks changes made to code over time. This ensures that multiple contributors can work on a project simultaneously without conflicts,

and it allows developers to roll back to previous versions if needed.
In this course, we will use GitHub to share code and resources. You can find a lot of programming projects on GitHub, and you can also share your own projects with others. If you are new to GitHub, you can find a lot of tutorials and guides on the internet. You can also find some useful links in the resources section.

2. GitHub Codespaces

GitHub Codespaces is a feature of GitHub that allows you to create a cloud-based development environment directly within GitHub. This means that you can write, run, and debug your code directly in your browser without having to install any software on your local machine. This is especially useful if you are working on a computer that does not have the necessary software installed, or if you want to work on your code from different devices. In this course, we will use GitHub Codespaces to write and run Python code and ensure that everyone has the same system basis. You can find more information about GitHub Codespaces in the resources section.

3. Visual Studio Code

The user interface of a Codespace is based on Visual Studio Code. Visual Studio Code is a very popular code editor and is used by many developers. It is open source and has a large community, so you can find many extensions and themes for it. It is developed by Microsoft for all systems. It is also very easy to use and has a lot of features that makes it a great choice for programming.

4. Jupyter Notebooks

Jupyter Notebooks are a great way to write and run Python code. They allow you to write code in cells and run them individually. This is very useful for testing small pieces of code or for writing code that is not part of a larger program. Jupyter Notebooks are widely used in the data science community and are a great tool for learning Python. You can find more information about Jupyter Notebooks in the resources section. In this course we will work sometimes with Jupyter Notebooks, but mostly we try to train you to work with "normal" Python scripts, so that you can use your knowledge in other environments as well.

Getting started

To start, all participants must set up a codespace. We will be working with codespaces throughout the course.

Follow the instructions below to open your codespace and dive into a hassle-free and efficient development experience. Enjoy the workshop!

1. Click on the `code` button at the top right
2. There you have two tabs: "Local" and "Codespaces". Click on "Codespaces"
3. Click on the Plus-Button to create a new codespace
4. A new window opens in which you can see a Visual Studio Code environment. This is your codespace. You can now start working on the content of the first learning unit in folder "session-1".

⚠️ Disclaimer ⚠️

This course is designed for beginners with no prior programming experience. We will cover the basics of Python programming and introduce you to some of the most common libraries used in the humanities. We will also provide you with some resources to help you continue learning on your own. **The most important thing is: do not be afraid to make mistakes, you can't break anything here.** Programming is a skill that takes time to develop, and the best way to learn is by doing. So don't be afraid to experiment and try new things. And also keep in mind that everyone has a different learning pace. So don't worry if you don't understand everything right away. Just keep practicing and most importantly: **have fun**!

Resources

Of course, this course can only show you the basics of Python programming. It's more of a chance to get started in the world of programming. You are strongly encouraged to continue to engage with these topics after this course, as this world has so much to offer and can make your life as a historian much easier. To help you get started, here is a small collection of further resources.

Python

- Python Documentation
- Python for Everybody
- Automate the Boring Stuff with Python
- Python Tutorials for Digital Humanities
- Fundamentals of Digital History
- Installing Python on your Computer:
  - https://wiki.python.org/moin/BeginnersGuide/Download
  - You can install a Windows Subsystem for Linux on your Windows computer. This allows you to use a fully functional Linux system (including terminal) under Windows -

    and much more interactively than with a virtual machine. Here are some instructions: https://learn.microsoft.com/en-us/windows/wsl/
  - https://programminghistorian.org/en/lessons/introduction-and-installation
  - Anaconda
    - Anaconda is a distribution of Python that comes with a lot of useful packages for data analysis and scientific computing. It also comes with a package manager called conda that makes it easy to install new packages. So it is a good choice for beginners where you have everything in one place. But it is very large and can take up a lot of space on your computer. We also recommend to get used to work with the "normal" Python installation.

Cleaning Data

- Regular Expressions:
  - Regular Expressions in Python
  - We have also prepared a small project to regular expressions in another course. Feel free to check it out.
- OpenRefine
  - OpenRefine is a powerful tool for cleaning and transforming messy data. It allows you to import data in various formats, clean it, transform it, and export it in various formats. It is especially useful for cleaning data that is not well-structured or that contains errors.
  - OpenRefine Tutorial
- Another typical tool for cleaning data is breaking down the whole text into its important parts. You can do this best with the Natural Language Toolkit (nltk) in Python.
  - We have also prepared a small project to nltk
- Large Language Model supported workflow for processing faulty OCR texts

GitHub and Git

- https://docs.github.com/en
- https://docs.github.com/en/repositories
- https://docs.github.com/en/codespaces
- https://git-scm.com/

General Resources

- Visual Studio Code
- Programming Historian

    - The Programming Historian offers novice-friendly, peer-reviewed tutorials that help humanists learn a wide range of digital tools, techniques, and workflows to facilitate their research.
- Jupyter Notebooks
    - The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations, and narrative text.
- We have offered a very similar course (in terms of content) where a few more learning resources can be found. For example, how to use the terminal or a few interesting example projects that are a good exercise.
- Virtual Environments in Python explained:
    - https://docs.python.org/3/library/venv.html
    - https://realpython.com/python-virtual-environments-a-primer/
    - https://docs.python-guide.org/dev/virtualenvs/
- HTML and CSS:
    - https://www.w3schools.com/html/
    - https://www.w3schools.com/css/
- Codecademy
    - Codecademy is an online interactive platform that offers free coding classes in 12 different programming languages including Python, Java, PHP, JavaScript, Ruby, SQL, and Sass, as well as markup languages HTML and CSS.

And some fun stuff

- Advent of Code
    - Advent of Code is an Advent calendar of small programming puzzles for a variety of skill sets and skill levels that can be solved in any programming language you like. People use them as a speed contest, interview prep, company training, university coursework, practice problems, or to challenge each other. It's a great way to practice your programming skills and have fun at the same time.

Session 1

Session 1: Introduction to Python programming

In this session, we will learn the very basics of Python. If you understand these concepts, you can build on them and learn more advanced concepts. So take your time to understand them! You will learn about:

- Data types
- Variables
- Operators that work with data types
- Lists, Tuples, and Dictionaries

How to work with Python

- **Python Interpreter**: You can run Python code in an interactive environment. You can start the Python interpreter by typing `python3` in your terminal. An interpreter is a special program that reads and executes code. You can type Python code directly into the interpreter and it will execute it immediately. This is a great way to test small pieces of code.
  - Try it out: Open your terminal and type `python3`. You should see the Python interpreter starting. Now you can type `print("Hello, World!")` and press enter. You should see the output `Hello, World!`.
- **Python Script**: You can write Python code in a file and run it with the Python interpreter. This is called a python script. You can create a new file with the extension `.py` and write your Python code in it. You can run the Python script by typing `python filename.py` in your terminal.
  - Try it out: Create a new file with the name `hello.py` and write the following code in it:

```python
print("Hello, World!")
```

Save the file and run it by typing `python3 hello.py` in your terminal. You should see the output `Hello, World!`.
To create a new file in Visual Studio Code, you can click on the `File` menu and then on `New File`. You can save the file by clicking on the `File` menu and then on `Save As...`.

- In Visual Studio Code, you can run Python code (if you have the Python extension installed) by pressing `Ctrl + Alt + N` or by clicking on the `Run Python File in Terminal` button ▶ in the top right corner of the editor.
  - Try it out: Open the file `hello.py` in Visual Studio Code and run it by pressing `Ctrl + Alt + N`. You should see the output `Hello, World!`.

Basic vocabulary

Before we start with the first Python program, we should clarify some basic terms. These are the basic vocabulary of Python:

- **Expression**: An expression is a combination of values, variables, operators, and calls to functions. Expressions need to be evaluated. In the example above, `number_1 + number_2` is an expression.
- **Value**: A value is one of the basic things a program works with. For example, the strings `"Hello, i am a Python program that adds two numbers."`, `"The first number is: "`, `"The second number is: "`, and `"The sum is: "` are all values.
- **Variable**: A variable is a container for a value. It can be used to store data. In the example above, `number_1` and `number_2` are variables. Variables are written without quotes. Variables are important because they allow you to store and reuse data instead of repeating yourself.
- **Operator**: An operator is a special symbol that represents a simple computation like addition, multiplication, or string concatenation. In the example above, `+` is an operator.
- **Data type**: A data type is a category for values, and every value belongs to exactly one data type. In the example above, the data type of `number_1` and `number_2` is an integer, the data type of the strings is a string.

Errors

When you write a program, you will make errors. Errors can be of different types. The most common types of errors are:

- **Syntax errors**: These are errors where the parser finds an incorrect statement. It will not execute the code. These are errors you will get when you make a typo or forget to add a closing bracket, for example.
- **Runtime errors**: These are errors that happen when the program is running. These are also called exceptions. These are errors you will get when you try to access a variable that is not defined, for example.
- **Semantic errors**: These are errors in logic. These are errors you will get when you write a program that is syntactically correct and runs, but does not do what you intended it to do.

Errors are normal and they are a part of the learning process. You should not be afraid of making errors. You should read the error messages carefully and try to understand what they are telling you. This is the best way to learn from it. So no worries!

Variables

You can store values in variables with an **assignment statement**. An assignment statement consists of a variable name, an equal sign (called the assignment operator), and the value to be stored. If you want to store the value `10` in the variable `number_1`, you can do this with the following code:

```python
number_1 = 10
```

Storing a value in a variable is often necessary. For example, you want to program a individual greeting message to your fellow students. You don't want to write every single greeting message by hand. For this, you can create a variable `name` and store the name of the person in it. Then you can use this variable in your greeting message.
You can also overwrite the value of a variable by assigning a new value to it. If you want to store the value `20` in the variable `number_1`, you can do this with the following code:

```python
number_1 = 10
print(number_1) # Output: 10
number_1 = 20
print(number_1) # Output: 20
```

The naming of a variable is very important. You should always choose a name that describes the content of the variable. The name of a variable can contain letters, numbers, and underscores. It must start with a letter or an underscore. It is case-sensitive, so `number_1` and `Number_1` are two different variables.

```python
Number_1 = 10
print(number_1) # Output: 10
number_1 = 20
print(Number_1) # Output: 10
```

You can work with variables as you would with values. You can use them in expressions, and you can pass them to functions. In the example above, we used the variables `number_1` and `number_2` in the expression `number_1 + number_2`.

Data types

- **Integers**: Integers (ints) are whole numbers. In the example above, `10` and `20` are integers.
- **Floating-point numbers**: Floating-point numbers (floats) are numbers with a decimal point. For example, `1.43` and `10.0` are floats.
- **Strings**: Strings (str) are sequences of characters. In the example above, `"Hello, i am a Python program that adds to numbers."`, `"The first number is: "`, `"The second number is: "`, and `"The sum is: "` are all strings.

- **Booleans**: Booleans (bools) are either `True` or `False`. In the example above, `True` and `False` are booleans.

Operators

Operator	Operation	Example	Result
`+`	Addition	`2 + 2`	`4`
`-`	Substraction	`5 - 2`	`3`
`*`	Multiplication	`5 * 2`	`10`
`/`	Division	`14 / 4`	`3.5`
`//`	Integer division/floored quotient	`14 // 4`	`3`
`%`	Modulus	`25 % 7`	`4`
`**`	Exponent	`2 ** 4`	`16`

Comments

You can add comments to your code. Comments are ignored by the Python interpreter. They are used to explain what the code does. You can add a comment by using the `#` symbol. Everything after the `#` symbol is ignored by the Python interpreter.

```python
# This is a comment
print("Hello, World!") # This is also a comment
```

Python Built-in Functions

Python has many built-in functions. These functions are always available, so you can use them in your programs directly. Some of the most important built-in functions are:

- `print()`: Prints the given object to the standard output device (screen) or to the text stream file.
- `input()`: Reads a line from input, converts it to a string (stripping a trailing newline), and returns that.
- `len()`: Returns the length (the number of items) of an object.
- `type()`: Returns the type of the object.
- `int()`: Returns an integer object from any number or string.
- `float()`: Returns a floating-point object from any number or string.
- `str()`: Returns a string object from any number or string.

Lists and Tuples

To store multiple values in one variable, you can use lists and tuples. A list is a collection which is ordered and **changeable**. In Python lists are written with square brackets. You have many possibilities to work with lists. You can add, remove, or change elements.

Example list:

```python
fruits = ["apple", "banana", "cherry"]
print(fruits) # Output: ["apple", "banana", "cherry"]
```

Tuples are almost identical to lists, but written with parentheses instead of square brackets. The main difference is that tuples are **immutable**. This means that you can't change the elements of a tuple once it has been assigned. This is good for storing values that should not be changed, such as days of the week or dates on a calendar.

Example tuple:

```python
fruits = ("apple", "banana", "cherry")
print(fruits) # Output: ("apple", "banana", "cherry")
```

Dictionaries

A dictionary is a collection which is unordered, changeable, and indexed. In Python dictionaries are written with curly brackets, and they have keys and values. You can access the items of a dictionary by referring to its key name, inside square brackets.

```python
# Dictionary
person = {
  "name": "Sara",
  "language": "python",
  "country": "Germany"
}
print(person) # Output: {'name': 'Sara', 'language': 'python', 'country':
'Germany'}
```

You have many possibilities to work with dictionaries. You can add, remove, or change elements.

```python
# Add an item to a dictionary
person["age"] = 25
print(person) # Output: {'name': 'Sara', 'language': 'python', 'country':
'Germany', 'age': 27}
```

```python

# Remove an item from a dictionary
person.pop("age")
print(person) # Output: {'name': 'Sara', 'language': 'python', 'country': 'Germany'}

# Change an item in a dictionary
person["country"] = "USA"
print(person) # Output: {'name': 'Sara', 'language': 'python', 'country': 'USA'}

# Get the value of a specific key
print(person["name"]) # Output: Sara

# Get all keys of a dictionary
print(person.keys()) # Output: dict_keys(['name', 'language', 'country'])

# Get all values of a dictionary
print(person.values()) # Output: dict_values(['Sara', 'python', 'USA'])

# Get all items of a dictionary
print(person.items()) # Output: dict_items([('name', 'Sara'), ('language', 'python'), ('country', 'USA')])

# Check if a key exists in a dictionary
if "name" in person:
  print("Yes, 'name' is one of the keys in the person dictionary")

# Or more specific
if "name" in person.keys():
  print("Yes, 'name' is one of the keys in the person dictionary")

if "python" in person.values():
  print("Yes, 'python' is one of the values in the person dictionary")

```

Session 2

Session 2: Working with Python Libraries and functions

You have learned some of the basics of Python in the first session:

- Variables
- Data types
- Operators
- Lists

In this session you will learn more about Python and its possibilities. You will learn about:

- **Flow Control**: If-Elif-Else Statements, While Loops, For Loops
- **Strings**: How to work with strings in Python
- **Working with files**: How to read from and write to files
- **Functions**: How to define and use functions
- **Python Libraries**: How to use Python libraries

These topics are very important to structure your code and to make it more readable and reusable. You can find this concepts in nearly every programming language, so take your time to understand them.

Flow Control

Flow control statements are used to control the flow of execution in a program. At many points in your program, you may want to make a decision about which block of code to execute next. For example: You have an application that displays the weather of the day and a what clothing would be appropriate. If the temperature is below 10 degrees Celsius (`temperature < 10`), you want to display a message that it is cold and you should wear a coat. If the temperature is between 10 and 20 degrees Celsius (`temperature >= 10 and temperature <= 20`), you want to display a message that it is warm and you should wear a light jacket. If the temperature is above 20 degrees Celsius (`temperature > 20`), you want to display a message that it is hot and you should wear a t-shirt.
You can do this with flow control statements. For this you need comparison operators and especially boolean values.

Comparison Operators

Operator	Meaning
==	Equal
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Boolean operators

Operator	Meaning
and	Logical and
or	Logical or
not	Logical not

The and operator's truth table:

Expression	Evaluates to
True and True	True
True and False	False
False and True	False
False and False	False

The or operator's truth table:

Expression	Evaluates to
True or True	True
True or False	True
False or True	True
False or False	False

The not operator's truth table:

Expression	Evaluates to
not True	False
not False	True

If-Elif-Else Statements

if-statement: The `if` statement is used to check a condition. If the condition is true, a block of code will be executed. If the condition is false, the block of code will be skipped.

```
if 5 > 2:
  print("Five is greater than two!")

if month == "December" and day == 24:
  print("It's Christmas Eve!")

```

else-statement: The `else` statement is used to execute a block of code if the condition is false.

```
if 5 < 2:
  print("Five is less than two!")
else:
  print("Five is not less than two!")
```

elif-statement: The `elif` statement is used to check multiple conditions. If the condition is true, the block of code will be executed. If the condition is false, the next condition will be checked.

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

While Loops

A `while` loop is used to execute a block of code as long as a condition is true.

```
i = 1
while i < 6:
```

```python
  print(i)
  i += 1
```

For Loops

A `for` loop is used to iterate over a sequence (list, tuple, dictionary, set, or string).

```python
for i in range(6):
  print(i)

# range() is a built-in function that returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.
```

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  print(fruit)
```

Strings

Strings are sequences of characters. You can use strings to represent text data. Strings in Python are surrounded by either single quotation marks, or double quotation marks.

```python
# Single quotation marks
print('Hello')

# Double quotation marks
print("Hello")
```

You have many possibilities to work with strings. You can add, remove, or change elements.

```python
# String
a = "Hello, World!"

# Get the character at position 1
print(a[1]) # Output: e

# Substring. Get the characters from position 2 to position 5 (not included)
print(a[2:5]) # Output: llo

# The strip() method removes any whitespace from the beginning or the end
b = " Hello, World! "
print(b.strip()) # Output: Hello, World!
print(b.lstrip()) # Output: Hello, World!
print(b.rstrip()) # Output:  Hello, World!

# The lower() method returns the string in lower case
print(a.lower()) # Output: hello, world!

# The upper() method returns the string in upper case
print(a.upper()) # Output: HELLO, WORLD!

# The replace() method replaces a string with another string
print(a.replace("H", "J")) # Output: Jello, World!

# The split() method splits the string into substrings if it finds instances
of the separator
print(a.split(",")) # Output: ['Hello', ' World!']

# The len() method returns the length of a string
print(len(a)) # Output: 13

# The in operator checks if a substring is present in a string
print("Hello" in a) # Output: True

# The not in operator checks if a substring is not present in a string
print("Hello" not in a) # Output: False

# You can use the + operator to concatenate two strings
a = "Hello"
b = "World"
c = a + b
print(c) # Output: HelloWorld

# You can also use the string interpolation to concatenate two strings
c = "%s to the %s" % (a, b)
print(c) # Output: Hello to the World

# You can also use f-strings to concatenate two strings. This is the most
modern way to do it.
c = f"{a} to the {b}"
print(c) # Output: Hello to the World

# If you have a list of strings, you can concatenate them with the join()
method
fruits = ["apple", "banana", "cherry"]
print(", ".join(fruits)) # Output: apple, banana, cherry

```

Working with files

In Python, you can easily work with text files to read from or write to them.

The `with` Statement

The `with` statement in Python is used to open files safely and automatically handle closing them, even if an error occurs. This helps avoid leaving files open, which can cause issues, especially with large numbers of files.

```python
# Open a file for reading
with open("example.txt", "r") as file:
    content = file.read() # Reads the entire file content
```

In this example:

- `open("example.txt", "r")` opens the file in read mode (`"r"`).
- The `with` statement ensures the file is automatically closed after the block.

Reading from a File

You can read a file in various ways, depending on your needs:

- **Read the whole file at once**:

```python
  with open("example.txt", "r") as file:
      content = file.read()
      print(content)
```

- **Read line by line**:

```python
  with open("example.txt", "r") as file:
      for line in file:
          print(line.strip())  # strip() removes newline characters
```

Writing to a File

To write data to a file, use the write mode (`"w"`) or append mode (`"a"` if you want to add to an existing file without overwriting it):

- **Overwrite or create a file**:

```python
  with open("example.txt", "w") as file:
    file.write("Hello, world!\n")
    file.write("Writing to a file is easy in Python.")
```

- **Append to an existing file**:

```python
with open("example.txt", "a") as file:
    file.write("\nAdding a new line to the file.")
```

> **Note**: Using `"w"` will overwrite the file if it exists, while `"a"` will add to it without overwriting.
> s

Functions

Functions are a block of code that only runs when it is called. You can pass data, known as parameters or arguments, into a function and the function can return data as a result. You can compare it with a recipe. You have a set of instructions and you can use it whenever you want. You already used some functions, like `print()` or `len()`.
Functions are part of every programming language and are very important to structure your code. They make your code more readable and reusable, because you can use the same code multiple times and you don't have to write it again.

A function has the following elements:

- The def keyword
- A function name
- Argument(s) (optional)
- A docstring to explain your function (optional)
- Function body
- A return statement (optional)

The function then looks like this:

```python
def my_function(name):
  """
  This function prints the name.
  """
  greeting = f"Hello, {name}!"
  return greeting
```

This function is called `my_function`. In Python you start with the keyword `def`, which means you define a function. Then you write the function name. After the function name you can define arguments in parentheses. In this case the function takes one argument, `name`, and returns a greeting. You can call this function with the following code:

```python
print(my_function("Alice")) # Output: Hello, Alice!
```

You can also define a function without arguments and without a return statement:

```python
def my_function():
  """
  This function prints a greeting.
  """
  print("Hello, World!")
```

It would look like this:

```python
my_function() # Output: Hello, World!
```

Of course you can also define a function with multiple arguments:

```python

def add_numbers(number1, number2):
  """
  This function adds two numbers.
  """
  return number1 + number2
```

The function `add_numbers` takes two arguments, `number1` and `number2`, and returns the sum of these two numbers. The arguments are separated by a comma. You can name them as you like. They are just placeholders for the values you pass to the function, you could also name them `a` and `b` for example. To call this function you can use the following code:

```python
print(add_numbers(5, 3)) # Output: 8
```

Sometimes it should be optional to pass an argument to a function. You can define a default value for an argument. If you don't pass a value to the function, the default value will be used.

```python
def my_function(name="World"):
  """
  This function prints a greeting.
  """
  print(f"Hello, {name}!")
```

The result would be:

```python
my_function() # Output: Hello, World!
my_function("Alice") # Output: Hello, Alice!
```

An argument doesn't have to be a string. It can be any data type, like a number, a list, a dictionary, or a boolean. You can also pass a function as an argument to another function. This is called a higher-order function.

```python
def my_function(names):
  """
  This function prints a list of names.
  """
  for name in names:
    print(name)
```

You can call this function with a list of names:

```python
names = ["Alice", "Bob", "Charlie"]
my_function(names)
```

Libraries

Libraries

A library in Python is a collection of packages and modules bundled together, usually designed for a specific purpose or task. Libraries make it easy to perform complex tasks without needing to write all the code yourself. For example, a fun library for us is *wikipedia*. With this package, you can search for Wikipedia articles and display their content directly in Python. This is great for quick research or to look up historical events without leaving your code. You first install it and then import it into your script.
To install it with pip, you can use the following command in your terminal:

```
pip install wikipedia
```

Then you are ready to use this library in your code:

```python
import wikipedia

# Get a summary of a Wikipedia article
summary = wikipedia.summary("witch hunts")
print(summary)
```

This will print a short summary of the Wikipedia article about "witch hunts". You can, of course, try out other search terms and quickly access information directly from your Python code.

Libraries like this are widely used to extend Python’s functionality for specific tasks, such as data processing, web development, and scientific computing. You can find nearly any library you need on the Python Package Index (PyPI).

⚠️ **NOTE** ⚠️ Never install a package on your system without using a virtual environment. In our case you don't need to use virtual environments, because we work in a GitHub Codespace. But it is good to know what they are and how to use them for your own (and local) projects. See below for more information about virtual environments.

---

pip

pip is Python's package manager, which makes it easy to install and manage libraries from the Python Package Index (PyPI). With `pip`, you can install libraries and packages from the command line or terminal without manually downloading files. For example, to install the `wikipedia` library, you would use:

```
pip install wikipedia
```

After running this command, you can import and use `wikipedia` in your code. `pip` also allows you to uninstall packages or check which ones are installed with commands like `pip uninstall` and `pip list`.

Documentation

When working with libraries, it is important to read the documentation to understand how to use them. The documentation provides information on the library's functions, classes, and methods, as well as examples and best practices. You can find the documentation for most libraries on their official websites or on the Python Package Index (PyPI). You should get used to reading documentation and using it as a reference when working with new libraries.
Also for your own code it is good practice to write documentation. This makes it easier for others (and yourself) to understand your code. You can write documentation using comments in your code, or by using docstrings (multi-line comments) at the beginning of your functions and classes. This is especially important when working on larger projects or collaborating with others.

💡 Virtual Environments 💡

In our case you don't need to use virtual environments, because we work in a GitHub Codespace. But it is good to know what they are and how to use them for your own (and local) projects.

Virtual Environments allow you to create isolated Python environments for different projects, so that each environment has its own set of packages and dependencies. This is useful when working on multiple projects that may require different versions of the same libraries. To create a virtual environment, navigate to your project folder in the terminal and run:

```
python -m venv myenv
```

This creates a virtual environment named `myenv`. But you can name it whatever you want, one common name is `.venv`.
To activate it, use:

- On Windows: `myenv\Scripts\activate`
- On macOS/Linux: `source myenv/bin/activate`

Once activated, any packages you install with `pip` will be specific to that environment. To exit the virtual environment, simply type `deactivate`.

Session 3

Session 3: Techniques for text analysis

You have learned the following concepts in Python so far:

- **Variables**: How to store and manipulate data
- **Data types**: Different types of data in Python (e.g., strings, integers, lists, dictionaries)
- **Control structures**: How to use `if` statements and loops
- **Strings**: How to work with strings in Python
- **Working with files**: How to read from and write to files
- **Functions**: How to define and use functions
- **Python Libraries**: How to use Python libraries

In this session you will learn some basic techniques for text analysis using Python:

- **Natural Language Processing (NLP)**: An introduction to NLP and its applications
- **Named Entity Recognition (NER)**: How to identify and classify named entities in a text
- **Text Preprocessing**: Techniques for cleaning and preparing text data for analysis
- **Word Frequency Analysis**: How to analyze the frequency of words in a text
- **N-gram Analysis**: How to analyze sequences of words in a text
- **TF-IDF (Term Frequency-Inverse Document Frequency)**: A statistical measure used to evaluate the importance of a word in a document relative to a collection of documents
- **Libraries for Text Analysis Tasks**: An overview of popular Python libraries for text analysis, including SpaCy, NLTK, and Gensim

This concepts are a good toolbox for historians to analyze texts and extract meaningful information from them. They are powerful techniques that can be applied to a wide range of historical texts, from letters and diaries to newspapers and official documents. For now we will work with a text from Edgar Allan Poe to practice these techniques. After this session you should be able to apply these techniques to the Salem Witch Trials and think about a research question you want to explore with text analysis.

Basic vocabulary

Corpus: A corpus is a large collection of texts that are used for linguistic analysis and research. In the context of text analysis, a corpus can be made up of various types of texts, such as books, articles, speeches, or social media posts. The purpose of a corpus is to provide a representative sample of language use in a particular domain or context, which can then be analyzed to identify patterns, trends, and other linguistic features. For example, our Salem

Witchcraft Papers could be considered a corpus of historical texts related to the Salem Witch Trials.

Document: A document is a single piece of text that is part of a larger corpus. In the context of text analysis, a document can be any type of written or spoken material, such as a book, article, email, or speech. Documents are typically analyzed individually or in relation to other documents in the corpus to identify patterns and trends in language use. For example, a letter written by a person accused of witchcraft during the Salem Witch Trials could be considered a document within the larger corpus of texts related to the trials.

Token: A token is a single unit of text that is used in natural language processing (NLP) and text analysis. Tokens can be words, phrases, or even individual characters, depending on the level of analysis being performed. In most cases, tokens are created by breaking down a larger piece of text into smaller, more manageable units. For example, the sentence "The quick brown fox jumps over the lazy dog" can be tokenized into individual words: ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog"]. Tokenization is an important step in many NLP tasks, such as text classification, sentiment analysis, and named entity recognition.

Stop words: Stop words are common words that are often removed from text data during natural language processing (NLP) and text analysis. These words are typically considered to be of little value in terms of meaning or context, and their removal can help to reduce noise and improve the accuracy of analysis. Examples of stop words in English include "the", "is", "in", "and", "to", and "a". However, the specific list of stop words can vary depending on the language, domain, and context of the text being analyzed.

Bag of Words (BoW): The Bag of Words (BoW) model is a simple and commonly used technique in natural language processing (NLP) and text analysis. In this model, a piece of text (such as a document or sentence) is represented as a "bag" of its individual words, without considering the order or context in which they appear.

Machine Learning: Machine learning is a subset of artificial intelligence (AI) that focuses on the development of algorithms and statistical models that enable computers to learn from and make predictions or decisions based on data. In the context of natural language processing (NLP) and text analysis, machine learning techniques are often used to build models that can automatically classify, cluster, or extract information from text data.

Some important concepts

Natural Language Processing (NLP)

Natural Language Processing (NLP) is a field of artificial intelligence that focuses on the interaction between computers and human (natural) languages. It involves the development of algorithms and models that enable computers to understand, interpret, and generate human language in a way that is meaningful and useful. NLP has a wide range of applications, including:

- Text classification (e.g., spam detection, sentiment analysis)
- Named entity recognition (NER)
- Machine translation (e.g., Google Translate)
- Chatbots and virtual assistants (e.g., Siri, Alexa)
- Text summarization
- Speech recognition
- and many more.

For your historical research, NLP can help you analyze large volumes of text data, extract relevant information, and uncover patterns and trends that may not be immediately apparent through manual analysis. For example, you can use NLP to extract names of people, places, and events from historical documents, analyze the sentiment of letters and diaries, or identify common themes and topics in newspapers and official records.

Named Entity Recognition (NER)

Named Entity Recognition (NER) is a subtask of NLP that involves identifying and classifying named entities in a text into predefined categories such as persons, organizations, locations, dates, and more. NER is useful for extracting structured information from unstructured text and can help historians identify key entities and their relationships in historical documents. For example, in a historical text, NER can help identify the names of historical figures, places mentioned, dates of events, and organizations involved. This information can then be used to create timelines, maps, and social networks, or to analyze the roles and relationships of different entities in historical events.

NER - same as NLP - is based on machine learning models that have been trained on large datasets of annotated text. These models learn to recognize patterns in the text and can then be applied to new, unseen texts to identify named entities.

Text Preprocessing

Text preprocessing is a crucial step in NLP that involves cleaning and transforming raw text data into a format that is suitable for analysis. Common text preprocessing techniques include:

- Tokenization: Splitting text into individual words or tokens.
- Stemming: Reducing words to their root form (e.g., "running" to "run").
- Lemmatization: Reducing words to their base or dictionary form (e.g., "better" to "good").
- Removing stop words: Eliminating common words that do not carry significant meaning (e.g., "the", "is", "and").

- Lowercasing: Converting all text to lowercase to ensure uniformity.
- Removing punctuation and special characters.

This is important because raw text data can be noisy and inconsistent, and preprocessing helps to standardize the text and reduce its complexity. Even if it seems unusual to convert a text into a form that is more difficult for humans to read, it is necessary for computers to process and analyze the text effectively, especially because you want to focus on the meaning of the words and not on their specific forms.

Word Frequency Analysis

Word frequency analysis involves counting the occurrences of each word in a text or a collection of texts. This technique helps identify the most common words and can provide insights into the main themes and topics of the text. For example, in a historical text, word frequency analysis can help identify key terms and concepts that are frequently mentioned, such as "witch", "trial", "accusation", etc. This information can be used to create word clouds, frequency distributions, and other visualizations that highlight the most important words in the text. And often it reveals interesting patterns, e.g. if certain words are used more frequently in specific contexts or time periods.

N-gram Analysis

N-gram analysis involves examining sequences of 'n' consecutive words (n-grams) in a text. For example, bigrams (2-grams) are pairs of consecutive words, while trigrams (3-grams) are triplets of consecutive words. N-gram analysis can help identify common phrases and patterns in the text. For example, in the sentence "The quick brown fox jumps over the lazy dog", the bigrams would be:

- "The quick"
- "quick brown"
- "brown fox"
- "fox jumps"
- "jumps over"
- "over the"
- "the lazy"
- "lazy dog"

This technique shows how words are used in context and can help identify collocations (words that frequently occur together) and common expressions in the text. For historians, n-gram analysis can reveal important phrases and terminology used in historical documents, which can provide insights into the language and discourse of the time. Compared to word frequency analysis, n-gram analysis provides more context about how words are used together, which can be particularly useful for understanding idiomatic expressions and specific terminology in historical texts.

TF-IDF (Term Frequency-Inverse Document Frequency)

TF-IDF is a statistical measure used to evaluate the importance of a word in a document relative to a collection of documents (corpus). It combines two metrics:

- Term Frequency (TF): The number of times a word appears in a document.
- Inverse Document Frequency (IDF): A measure of how common or rare a word is across all documents in the corpus.

TF-IDF helps identify words that are significant in a specific document while down-weighting common words that appear frequently across all documents. Compared to simple word frequency analysis or n-gram analysis, TF-IDF provides a more nuanced view of word importance by considering both the frequency of a word in a document and its rarity across the corpus. This makes TF-IDF particularly useful for tasks such as information retrieval, document classification, and topic modeling, where the goal is to identify words that are most relevant to a specific document or topic.

Topic Modeling

Topic modeling is a technique used to identify and extract topics from a collection of documents. It is a machine learning method that analyzes the words in the documents and groups them into topics based on their co-occurrence patterns. Topic modeling can help to identify underlying themes and topics in large collections of historical texts, such as letters, diaries, newspapers, and official records. By analyzing the topics present in the texts, historians can gain insights into the main issues and concerns of a particular time period or social group. Topic modeling can also be used to track changes in topics over time, identify relationships between different topics, and explore the connections between different documents in a corpus.

Document Clustering

Document clustering is a technique used to group similar documents together based on their content. It is an unsupervised machine learning method that analyzes the words and phrases in the documents and identifies patterns of similarity.

Sentiment Analysis

Sentiment analysis is a technique used to determine the emotional tone or sentiment expressed in a piece of text. It is a form of natural language processing (NLP) that involves analyzing the words and phrases in the text to identify whether the sentiment is positive, negative, or neutral. For example, you can use sentiment analysis to analyze letters, diaries, or newspaper articles from a particular time period to understand the emotional tone of the texts. This can provide insights into the attitudes and opinions of individuals or groups during that time period, as well as the social and political context in which the texts were written.

Which to use for your research?

The choice of technique depends on your specific research question and the nature of the text data you are analyzing. Most of the time you will use a combination of these techniques to gain a comprehensive understanding of the text data. For example, you might start with text preprocessing to clean and prepare the data, then use word frequency analysis and n-gram analysis to identify common words and phrases, and finally apply TF-IDF or topic modeling to identify important words and topics in the text. You can also use named entity recognition (NER) to extract specific entities such as names, places, and dates from the text.

Always keep in mind that these techniques are tools to help you analyze and interpret the text data, but they should be used in conjunction with your own knowledge and expertise as a historian. It is important to critically evaluate the results of your analysis and consider the historical context in which the texts were written. Furthermore, most of the methods are not specialised for historical texts, so you might need to adapt them to the specific characteristics of your data. Today we will focus on NER, text preprocessing, word frequency analysis, n-gram analysis and TF-IDF.

Libraries for Text Analysis Tasks

spaCy

SpaCy is an open-source library for advanced NLP in Python. It provides pre-trained models and tools for various NLP tasks, including tokenization, part-of-speech tagging, named entity recognition (NER), dependency parsing, and more. SpaCy is designed for performance and ease of use, making it a popular choice for NLP applications.

NLTK

The Natural Language Toolkit (NLTK) is a comprehensive library for NLP in Python. It offers a wide range of tools and resources for text processing, including tokenization, stemming, lemmatization, parsing, and classification. NLTK also provides access to various corpora and lexical resources, making it a valuable tool for researchers and developers working with text data.

Gensim

Gensim is a library for topic modeling and document similarity analysis in Python. It is particularly well-suited for handling large text corpora and provides efficient implementations of algorithms such as Latent Dirichlet Allocation (LDA) and Word2Vec. Gensim is widely used for tasks such as topic modeling, document clustering, and semantic similarity analysis.

Exercises

Session 3 — Techniques for Text Analysis (for Historians)

Exercise 1 — Text Preprocessing with NLTK

We will start with a simple text preprocessing procedure using the NLTK library. The goal is to prepare the text by tokenizing, stemming, lemmatizing, and removing stopwords.

NLTK (Natural Language Toolkit) is a powerful library for natural language processing in Python. It provides easy-to-use interfaces to over 50 corpora and lexical resources, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning. You can have a look at the official NLTK documentation for more information. There you find installation instructions, tutorials, and guides on how to use the library.

1. First of all, we want to install the NLTK library. Type the following command in your terminal:

```
pip install --user -U nltk
```

After the installation process, NLTK is ready to use in your codespace.

2. Now you can open the file exercise_scripts/exercise_1.py. There you need to import the necessary NLTK modules and download the required resources (like punkt and stopwords). You can do this by adding the following lines at the beginning of your script:

```python
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

nltk.download('punkt_tab')
nltk.download('wordnet')
nltk.download('stopwords')
```

First you need to import the NLTK library itself to access all its functions. Then, you import the `stopwords` module from the `nltk.corpus` package to access the list of stopwords. Finally, you import the `WordNetLemmatizer` class from the `nltk.stem` package to perform lemmatization. The `nltk.download()` function is used to download the necessary resources for tokenization, lemmatization, and stopword removal.

3. Next, we want to access our example text files in the `data` folder. With digital methods, you typically work with a large number of documents, not just one. In the last exercises, we always called up the texts individually. If you are writing programs, you don't want to repeat yourself. We want to read in all texts in a folder and store them in a list. For this, we want to use the `os` module, which provides a way of using operating system-dependent functionality like reading files from directories. This step is necessary, because you need the right path to access the files in the `data` folder. The `os` module helps you to create these paths.

You can use the following code snippet to read all text files from the `data` folder and store their contents in a list:

```python
import os # Import the os module at the beginning of your script

...

folder_path = './session_3/data/' # Path to the data folder
all_texts = [] # Empty list to store the texts
for filename in os.listdir(folder_path): # Loop through all files in the folder
  with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file: # Open each file with the open() function
      all_texts.append(file.read()) # Read the content and append it to the list

```

This code will read all `.txt` files in the specified folder and store their contents in the `all_texts` list. Add it to your code and create a new variable `our_text`. We want to start with only one text for now. You can choose any text from the `all_texts` list.

Here is how you can do it:

```python
our_text = all_texts[0]
```

You can play around with the index to access different texts in the list. For example, `our_text[0]` will give you the first text, `all_texts[1]` the second text, and so on.

4. Now we can work with the text. To analyse the text, we need to preprocess it first. The preprocessing steps include:

- Lowercasing the text
- Tokenizing the text into words
- Lemmatizing the tokens

- Removing stopwords

Let's start with lowercasing the text. You can do this by using the `.lower()` method on the text variable. You can find more information about this method in session 2. Save the result in a variable called `text`. Print the content of the variable to see if it worked.

```
text = our_text.lower()
```

You should now see the entire text in lowercase letters in your terminal. Before proceeding to the next step, make sure to comment out or remove the print statement to avoid cluttering your output in the following steps.

5. Next, we will tokenize the text into words. You can use the `nltk.word_tokenize()` function for this. Pass the lowercased text to this function (which means, you should write the variable name `text` in the parentheses) and save the result in a variable called `tokens`. Print the content of the variable to see if it worked.

```
tokens_full = nltk.word_tokenize(text)
```

6. Now we have a list of tokens, but it also contains punctuation and special characters. We want to keep only the alphabetic tokens. You can filter out non-alphabetic tokens. The method `.isalpha()` can be used to check if a token is alphabetic. Update the `tokens` variable to contain only alphabetic tokens. It should look like this:

```
tokens = []

for token in tokens_full:
    if token.isalpha():
        tokens.append(token)

# or the shorter version with list comprehension:
tokens = [token for token in tokens_full if token.isalpha()]
```

Print the content of the variable to see if it worked. Before proceeding to the next step, make sure to comment out or remove the print statement to avoid cluttering your output in the following steps.

7. Next, we will lemmatize the tokens using the WordNet Lemmatizer. First, create an instance of the `WordNetLemmatizer` class. This means you need to write `lemmatizer = WordNetLemmatizer()` to create a lemmatizer object with which you can lemmatize words, like dogs to dog, running to run, etc. This part of code should also be placed after the

`nltk.download('stopwords')` line. It is a good practice to put all the imports and initializations at the beginning of your script.

Next, create a new empty list called `lemmas` at the end of your script. Then, use a for loop to iterate over the `tokens` list and apply the `lemmatizer.lemmatize()` method to each token. Append the lemmatized tokens to the `lemmas` list. Print the content of the `lemmas` variable to see if it worked.

This should look like this:

```
lemmatizer = WordNetLemmatizer()

...

lemmas = []
for token in tokens:
    lemma = lemmatizer.lemmatize(token)
    lemmas.append(lemma)
```

Now you should see the lemmatized tokens in your terminal. Before proceeding to the next step, make sure to comment out or remove the print statement to avoid cluttering your output in the following steps.

8. Finally, we will remove stopwords from the lemmatized tokens. You can use the `stopwords.words('english')` function to get a list of English stopwords. Save `stopwords.words('english')` in a variable called `stop_words`.

Create a new empty list called `clean_tokens`. Then, use a for loop to iterate over the `lemmas` list and check if each token is not in the stopwords list. If it is not a stopword, append it to the `clean_tokens` list. Print the content of the `clean_tokens` variable to see if it worked.

This should look like this:

```
stop_words = stopwords.words('english')
clean_tokens = []
for lemma in lemmas:
    if lemma not in stop_words:
        clean_tokens.append(lemma)
```

You should now see the list of clean tokens without stopwords in your terminal. If you are interested, you can print out the stopword list with:

```
print(stop_words)
```

You may notice that the list contains common words like "the", "is", "in", etc. These words are often removed in text analysis because they do not carry significant meaning. But the list is for modern English texts. Since we are working with historical texts, some words might not be relevant for our analysis.

9. Sometimes you want to add custom stopwords that are specific to your text. You can create a list of custom stopwords and extend the existing stopwords list with it. For example:

```python
custom_stopwords = ['said', 'one', 'like'] # Add your custom stopwords here
stop_words = stopwords.words('english') # Get the default stopwords
stop_words.extend(custom_stopwords) # Extend the stopwords list with custom stopwords
```

Try it out by adding some words that you think are not useful for the analysis of our example text. Create the `custom_stopwords` list and extends your existing `stop_words` list with it before the for loop where you create the `clean_tokens` list. Then, run the code again and see how the `clean_tokens` list changes.

10. As a last step, we want to make our code more reusable. Therefore, we will wrap the preprocessing steps into a function called `preprocess_text()`. This function should take a text string `raw_text` as input and return the list of clean tokens. You can copy your code from the previous steps and paste it into the function. Make sure to indent the code correctly so that it is part of the function. The function should look like this:

```python
def preprocess_text(text):
    # Your preprocessing code here
    return clean_tokens
```

The steps of the functiions should be the following:

- Lowercase the input text
- Tokenize the text into words
- Filter out non-alphabetic tokens
- Lemmatize the tokens
- Remove stopwords (including custom stopwords)

Finally, call the function with `our_text` as an argument and save the result in a variable called `processed_text`. Print the content of the `processed_text` variable to see if it worked.

11. With the function in place, you can easily preprocess any text by calling `preprocess_text(your_text_here)`. You can also use this function with multiple texts if

you want to preprocess more than one document, as you will need to do later with the trial sources. Let's practice this.

Save your second text file in the variable second_text: `second_text = all_texts[1]`. Then, call the `preprocess_text()` function with `second_text` as an argument and save the result in a variable called `processed_second_text`. Print the content of the `processed_second_text` variable to see if it worked. Finally, use a for-loop to iterate over the texts list, preprocess each text using the preprocess_text() function, and print the result.

```
for text in all_texts:
    processed = preprocess_text(text)
    print(processed)
```

You can play around with the terminal outcome to make it more readable:

```
for text in all_texts:
    processed = preprocess_text(text)
    print("----- New Document -----")
    print(processed)
    print("\n")
    print("----- End Document -----")
```

Perfect! You have successfully completed the text preprocessing exercise using NLTK. You can now use the `preprocess_text()` function to preprocess any text data you want to analyze further.

You have learned:

- How to install and import the NLTK library
- How to read multiple text files from a folder
- How to preprocess text data by lowercasing, tokenizing, lemmatizing, and removing stopwords
- How to create a reusable function for text preprocessing

Exercises

Session 3 — Techniques for Text Analysis (for Historians)

Exercise 2 — Named Entity Recognition (NER) with SpaCy and Basic Co-Occurrence Analysis

In this exercise, we will use the SpaCy library to identify and analyze named entities such as persons, locations, and organizations in a text by Edgar Allan Poe.

After extracting these entities, we will extend our analysis with a basic co-occurrence approach that examines which words most often appear in the context of certain persons.

SpaCy is a powerful NLP library that provides pre-trained models for various languages and tasks, including NER. We will use SpaCy's English language model to process the text and extract named entities. You can find more information about SpaCy and its NER capabilities in the official documentation . There you can find installation instructions , guides , and a spaCy 101 .

1. First, we need to install the SpaCy library and download the English language model. You can do this by running the following commands in your terminal:

```
pip install -U pip setuptools wheel
```

```
pip install spacy
```

```
python -m spacy download en_core_web_md
```

This steps will install necessary packages, spaCy itself, and the medium-sized English language model. You can also choose to download the small or large models. If you want, you can play around with the different models later to see how they perform on the text. In our case, the medium model is a good balance between speed and accuracy, while the small model is faster but much less accurate, and the large model is more accurate but slower. To install the other models, you can use the following commands:

```
python -m spacy download en_core_web_sm  # Small model
python -m spacy download en_core_web_lg  # Large model
```

2. Next, we will import the necessary module and load the English language model in our script. Open the file `exercise_scripts/exercise_2.py` and add the imports at the beginning of your script:

```
import spacy
```

Then load the SpaCy model by adding the following line of code after the import:

```
nlp = spacy.load("en_core_web_md")
```

This will load the medium English language model which is sufficient for our NER task. You can also use the small model ("en_core_web_sm") if you want a quicker setup, but the medium model generally provides better accuracy. If you want, you can play around with different models and see how they perform on the text later:

```
nlp = spacy.load("en_core_web_sm")  # Small model
nlp = spacy.load("en_core_web_md")  # Medium model
nlp = spacy.load("en_core_web_lg")  # Large model
```

Remember to download the respective models using the command in your terminal:

```
python -m spacy download en_core_web_sm
python -m spacy download en_core_web_md
python -m spacy download en_core_web_lg
```

Now you can use the functionality of SpaCy in your code.

3. Next, we want to access our example text files in the `data` folder. With digital methods, you typically work with a large number of documents, not just one. In the last exercises, we always called up the texts individually. If you are writing programs, you don't want to repeat yourself. We want to read in all texts in a folder and store them in a list. For this, we want to use the `os` module, which provides a way of using operating system-dependent functionality like reading files from directories. This step is necessary, because you need the right path to access the files in the `data` folder. The `os` module helps you to create these paths.

You can use the following code snippet to read all text files from the `data` folder and store their contents in a list:

```
import os # Import the os module at the beginning of your script

...

folder_path = './session_3/data/' # Path to the data folder
all_texts = [] # Empty list to store the texts
for filename in os.listdir(folder_path): # Loop through all files in the
folder
  with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as
file: # Open each file with the open() function
      all_texts.append(file.read()) # Read the content and append it to the
list
```

This code will read all `.txt` files in the specified folder and store their contents in the `all_texts` list. Add it to your code and create a new variable `our_text`. We want to start with only one text for now. You can choose any text from the `all_texts` list.

Here is how you can do it:

`python   our_text = all_texts[4]`

You can play around with the index to access different texts in the list. For example, `our_text[0]` will give you the first text, `all_texts[1]` the second text, and so on. For this exercise, the first text (index 0) is the best choice to start with.

4. Now we can process the text using the SpaCy NLP pipeline. SpaCy marks named entities in the text and assigns them labels such as "PERSON" for persons, "GPE" for geopolitical entities (like cities or countries), and "ORG" for organizations. We can extract these entities and count how often they appear in the text. Here is an overview of the entity labels used by SpaCy:

Label	Description
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Organizations, companies, agencies, institutions, etc.
GPE	Countries, cities, states.
DATE	Absolute or relative dates or periods.
CARDINAL	Numerals that do not fall under another type.

Let's start by creating a function called `get_most_common_entities` that takes a text, an entity label (like "PERSON" or "GPE"), and a number `top_n` as input and returns the most common entities of that type in the text.

```
def get_most_common_entities(text, entity_label="PERSON", top_n=10):
```

The entity_label parameter uses "PERSON" as a default value, but you can call the function with other labels to get different types of entities. The top_n parameter specifies how many of the most common entities you want to return. As a default, we want to return the top 10 entities. Inside the function, we will process the text with SpaCy and extract the entities:

```
  doc = nlp(text)
  entities = []
  for ent in doc.ents:
      if ent.label_ == entity_label:
          entities.append(ent.text)
```

This code will create a `Doc` object by passing the text to the `nlp` object. In simple terms, this means that SpaCy will analyze the text and identify various linguistic features, including named entities. You don't need to worry about the details of how this works; just know that the `doc` object now contains all the information SpaCy has extracted from the text. Then, the code will iterate over the named entities in the `Doc` object and append the text of the entities that match the specified label to the `entities` list. Finally, we will count the occurrences of each entity and return the most common ones. For this, we will use the `Counter` class from the `collections` module. You need to import it at the beginning of your script:

```
from collections import Counter
```

Then, you can add the following code to the function to count the entities and return the most common ones:

```
  counts = Counter(entities)
  return counts.most_common(top_n)
```

The `most_common` method returns a list of the most common entities along with their counts.

Your function is now complete and should look like this:

```
def get_most_common_entities(text, entity_label="PERSON", top_n=10):
  doc = nlp(text)
  entities = []
```

```
    for ent in doc.ents:
        if ent.label_ == entity_label:
            entities.append(ent.text)
    counts = Counter(entities)
    return counts.most_common(top_n)
```

You can now call the function with `our_text` as an argument and print the result. You can also play around with the `entity_label` and `top_n` parameters to see how they affect the output. You can call the function like this:

```
top_persons = get_most_common_entities(our_text)
print("Most common persons:", top_persons)
```

Or to get the most common locations, you can call the function like this with the 3 top locations:

```
top_locations = get_most_common_entities(our_text, entity_label="GPE",
top_n=3)
print("Most common locations:", top_locations)
```

5. You may notice that some names appear multiple times in different forms (like "Mr. Smith" and "Smith"). To clean this up, you can create a merge function that combines these variations into a single name. You have different options for how to do this. In this exercise, we will use a simple approach. We will create a list with lists of name variations. For example:

```
merge_names = [
    ["Mr. Smith", "Smith", "John Smith"],
    ["Dr. Johnson", "Johnson"]
]
```

Remember that you have various options to work with lists. One is to create a list with lists in it. If you want to access the first list in the list, you can do it like this: `merge_names[0]`. If you want to access the first element of the first list, you can do it like this: `merge_names[0][0]`. You can also use a for loop to iterate over the lists and their elements. For example:

```
for name_list in merge_names:
    for name in name_list:
      print(name)
```

You can use this approach to do the following:

- Create a list `merge_names` with lists of name variations. The first name in each list will be the name that all variations will be merged into.
- Then we want to iterate over the `persons` list to check if a name is in one of the lists in `merge_names`. If it is, we will replace it with the first name of the corresponding list.
- With this approach, all variations of a name will be replaced with a single, consistent name.

So let's implement this in our `get_most_common_entities` function. You can add `merge_entity_names` as a parameter to the function with a default value of an empty list. Then we use a new concept: the `range` function. With this function, you can create a sequence of numbers. For example, `range(5)` will create a sequence of numbers from 0 to 4. You can use this function to iterate over the indices of the `persons` list. This way, you can access and modify the elements of the list directly. This code should take place before counting the names with `Counter`.

Here is how you can do it:

```
for entity_name_list in merge_entity_names: # Iterate over the lists of name variations
  for i in range(len(entities)): # Iterate over the indices of the persons list
      if entities[i] in entity_name_list: # Check if the name is in one of the lists
        entities[i] = entity_name_list[0] # Replace it with the first name in the corresponding list
```

This code will iterate over the indices of the `persons` list and check if each name is in one of the lists in `merge_names`. If it is, it will replace it with the first name in the corresponding list. Try to understand how this code snippet works. It may helps to print out single steps to see what is happening or maybe you can try to draw the steps on a piece of paper.

Your updated function should now look like this:

```
def get_most_common_entities(text, entity_label="PERSON", top_n=10, merge_entity_names=[]):
    doc = nlp(text)
    entities = []
    for ent in doc.ents:
        if ent.label_ == entity_label:
            entities.append(ent.text)

    # Merge name variations
    for entity_name_list in merge_entity_names:
        for i in range(len(entities)):
            if entities[i] in entity_name_list:
                entities[i] = entity_name_list[0]

    counts = Counter(entities)
    return counts.most_common(top_n)
```

Finally, call the function with a `merge_names` list and print the result to see if it worked:

```
merge_names = [["Augustus Bedloe", "Bedloe"], ["Warren Hastings", "Hastings"]]
most_common_entities = get_most_common_entities(our_text,
merge_entity_names=merge_names)
print(most_common_entities)
```

You may notice that "Bedloe" and "Augustus Bedloe" morph into "Augustus Bedloe" in the output. This changes the ranking of the most mentioned persons in the text and there is a new person in the top 10 list: "Augustus Bedlo". You should also add this name to the `merge_names` list. Play around with different name variations to see how they affect the output.

6. So far, we know who appears often.

Next, we want to explore which words typically occur around them — for example, what adjectives or nouns appear in the same sentence as the person. This can give us insights into how these persons are described or what actions they are associated with in the text. To do this, we will create a function called `get_entity_context` that takes a text, a list of entity names, and a number `top_n` as input and returns the most common words that appear in the context of those entities. The context is defined as the words that appear in the same sentence as the entity. Here is how you can implement this function:

First, you need to import the `Matcher` class from the `spacy.matcher` module at the beginning of your script:

```
from spacy.matcher import Matcher
```

Then, you can define the `get_entity_context` function like this:

```
def get_entity_context(text, entity_names, top_n=10):
```

Then we want to process the text with spaCy.

```
    doc = nlp(text)
    matcher = Matcher(nlp.vocab)
    for name in entity_names:
```

```
    pattern = [{"LOWER": token.lower()} for token in name.split()]
    matcher.add("NAME", [pattern])
```

This code will create a `Doc` object by passing the text to the `nlp` object. Then, it will create a `Matcher` object and add patterns for each entity name in the `entity_names` list. In detail, it works as follows:

- The `Matcher` class is used to find sequences of tokens that match specific patterns in the text. We create a `Matcher` object by passing the vocabulary of the `nlp` object.
- For each entity name in the `entity_names` list, we create a pattern that matches the lowercase version of the name. The pattern is a list of dictionaries, where each dictionary represents a token in the name. The `LOWER` key is used to match the lowercase version of the token. Or in very simple terms: We want to match the name regardless of whether it is written in uppercase or lowercase letters.
- Finally, we add the pattern to the matcher with the label "NAME". Or in simple terms: We tell the matcher to look for the patterns we just created.

Next, we will use the matcher to find the occurrences of the entity names in the text and extract the context words:

```
matches = matcher(doc)
context_words = []
for match_id, start, end in matches:
    span = doc[start:end].sent
    for token in span:
        if token.pos_ in ["NOUN", "VERB", "ADJ"]:
            context_words.append(token.lemma_.lower())
```

This code does the following:

- It uses the matcher to find all occurrences of the entity names in the `Doc` object and stores the matches in the `matches` variable.
- It initializes an empty list called `context_words` to store the context words.
- It iterates over the matches found by the matcher.
- For each match, it gets the sentence that contains the match using the `sent` attribute of the `Doc` object. `sent` is a property that returns the sentence span containing the matched entity.
- It then iterates over the tokens in the sentence and checks if the token is a noun, verb, or adjective using the `pos_` attribute of the token. The `pos_` attribute provides the part-of-speech tag of the token as a string.

- If the token is a noun, verb, or adjective, it appends its lemma (base form) in lowercase to the `context_words` list using the `lemma_` attribute of the token.

You don't need to worry about the details of how this works; just know that the `context_words` list now contains all the relevant words that appear in the same sentences as the specified entity names. If you want to learn more about the Matcher class and how it works, you can check out the official SpaCy documentation on the Matcher.

Finally, we will count the occurrences of each context word and return the most common ones:

```
    return Counter(context_words).most_common(top_n)
```

This code uses the `Counter` class from the `collections` module to count the occurrences of each context word in the `context_words` list. The `most_common` method returns a list of the most common context words along with their counts.

The function is now complete and should look like this:

```
def get_entity_context(text, entity_names, top_n=10):
  doc = nlp(text)
  matcher = Matcher(nlp.vocab)
  for name in entity_names:
    matcher.add("NAME", [[{"LOWER": name.lower()}]])

  matches = matcher(doc)
  context_words = []
  for match_id, start, end in matches:
      span = doc[start:end].sent
      for token in span:
          if token.pos_ in ["NOUN", "VERB", "ADJ"]:
              context_words.append(token.lemma_.lower())

  return Counter(context_words).most_common(top_n)
```

You can now call the function with `our_text` and a list of entity names as arguments and print the result. You can also play around with the `entity_names` and `top_n` parameters to see how they affect the output. For example, to get the 10 most common context words around the person "Templeton", you can call the function like this:

```
context_templeton = get_entity_context(our_text, ["Templeton"])
print("\nTop context words around 'Templeton':")
```

```
for word, freq in context_templeton:
    print(f"  {word}: {freq}")
```

This code will print the 10 most common context words that appear in the same sentences as "Templeton" along with their counts.

Play around with different names from the `most_common_entities` output to see what context words are associated with them. You can also try using multiple names at once by passing a list of names to the `entity_names` parameter, especially the merged names from the previous step:

```
context_multiple = get_entity_context(our_text, ["Augustus Bedloe", "Bedloe"],
top_n=10)

print("\nTop context words around 'Augustus Bedloe' and 'Bedloe':")

for word, freq in context_multiple:
    print(f"  {word}: {freq}")
```

In summary, you have learned:

- How to install and set up SpaCy for NER tasks.
- How to read and process text files using SpaCy.
- How to extract, filter, and count named entities.
- How to clean up entity names by merging variations.
- How to analyze the context of entities by finding co-occurring words in sentences.

These techniques can be applied to various types of texts and can provide valuable insights into the content and themes of the documents.

Exercises

Session 3 — Techniques for Text Analysis (for Historians)

Exercise 3 — Context Analysis with Word Frequency, N-Grams, and TF–IDF

In this exercise we perform a context analysis of one or more texts by identifying the most common words, extracting common word combinations (n-grams) using the NLTK library, and calculating TF–IDF scores using the Gensim library. These techniques help reveal the themes and topics present in the texts and highlight terms that are particularly characteristic of individual documents.

Gensim is a popular Python library for natural language processing, particularly known for its capabilities in topic modeling and document similarity analysis. You can find more information about Gensim and its documentation here: https://radimrehurek.com/gensim/ . The documentation contains useful tutorials and examples to help you get started with the library. We will use Gensim to calculate TF–IDF scores for words in our texts, but if you want to explore more advanced features, the documentation is a great resource.

Additionally, we will use the NLTK library to extract n-grams from the texts. NLTK (Natural Language Toolkit) is a widely used library for natural language processing tasks in Python. It provides tools for tokenization, stemming, lemmatization, and more. You can find more information about NLTK and its documentation here: https://www.nltk.org/ . The documentation includes tutorials and examples that can help you understand how to use the library effectively. You may notice, that it is a common practice to use multiple libraries together for different NLP tasks, as each library may have its own strengths and capabilities.

1. First, we need to install the Gensim library and the NLTK library. You can do this by running the following commands in your terminal:

```
pip install --upgrade gensim
```

```
pip install --user -U nltk
```

Now you have Gensim and NLTK installed in your Codespace and can use them in your Python scripts.

2. Now you can open the file `exercise_scripts/exercise_3.py`. In Exercise 1, we have preprocessed the text by lowercasing, tokenizing, lemmatizing, and removing stopwords. We will work with this preprocessed text for our context analysis. The file `exercise_3.py` already contains the preprocessing steps from Exercise 1, so you can start right there.

First, we want to access our example text files in the `data` folder. With digital methods, you typically work with a large number of documents, not just one. In the last exercises, we always called up the texts individually. If you are writing programs, you don't want to repeat yourself. We want to read in all texts in a folder and store them in a list. For this, we want to use the `os` module, which provides a way of using operating system-dependent functionality like reading files from directories. This step is necessary, because you need the right path to access the files in the `data` folder. The `os` module helps you to create these paths.

You can use the following code snippet to read all text files from the `data` folder and store their contents in a list:

```
import os # Import the os module at the beginning of your script below the other imports

...

folder_path = './session_3/data/' # Path to the data folder
all_texts = [] # Empty list to store the texts
for filename in os.listdir(folder_path): # Loop through all files in the folder
  with open(os.path.join(folder_path, filename), 'r', encoding='utf-8') as file: # Open each file with the open() function
      all_texts.append(file.read()) # Read the content and append it to the list
```

This code will read all `.txt` files in the specified folder and store their contents in the `all_texts` list. Add it to your code and create a new variable `our_text`. We want to start with only one text for now. You can choose any text from the `all_texts` list.

Here is how you can do it:

```
our_text = all_texts[0]
```

You can play around with the index to access different texts in the list. For example, `our_text[0]` will give you the first text, `all_texts[1]` the second text, and so on.

3. Now we have a text, but we need to preprocess it first. You can reuse the preprocessing steps from Exercise 1. For this, we need to take the `our_text` variable and apply the

preprocessing functions to it. It takes the text as input and returns a list of cleaned tokens.

Here is how you can do it:

```
clean_tokens = preprocess_text(our_text)
```

You can then print the `clean_tokens` variable to see the preprocessed tokens with `print(clean_tokens)`. What do you think about the result? If you see words that have no meaning, like "thee" or "thy", you can add them as custom stopwords to the function and rerun your script. For example:

```
custom_stopwords = ['thee', 'thy', 'thou'] # Add your custom stopwords here
clean_tokens = preprocess_text(our_text, custom_stopwords=custom_stopwords)
```

4. In this exercise we want to look at the most common words and their contexts. We will start by defining a function called `word_frequency` that takes a list of tokens as input and returns the most common words in the text. You can use the `Counter` class from the `collections` module to count the occurrences of each word. You can import it at the beginning of your script:

```
from collections import Counter
```

This module can be used right away without installation, as it is part of the Python standard library.

Now, we want to define the `word_frequency` function. It should take a list of tokens as an argument and a top_n parameter to specify how many of the most common words to return. Here is how you can implement it:

```
def word_frequency(tokens, top_n=10):
```

The list of tokens will be our preprocessed text `clean_tokens`. Inside the function, we can use the `Counter` class to count the occurrences of each word: `word_counts = Counter(tokens)`. And return the most common ones using the `most_common()` method: `word_counts.most_common(top_n)`. Here is the complete function:

```
def word_frequency(tokens, top_n=10):
    word_counts = Counter(tokens)
    return word_counts.most_common(top_n)
```

You can call this function with the `clean_tokens` variable and print the result to see the most common words in the text:

```python
common_words = word_frequency(clean_tokens, top_n=10)
print(common_words)
```

You may notice that some words appear very frequently. These words can give us an idea of the main themes and topics discussed in the text. If you think some of the words are not meaningful, you can add them to the custom stopwords list and rerun the preprocessing step from Exercise 3.3. Play around with the `top_n` parameter and see how many common words you want to display.

Before proceeding to the next step, make sure to comment out or remove the print statement to avoid cluttering your output in the following steps.

5. Next, we will define a function called `find_ngrams` that takes a list of tokens and an integer `n` as input and returns the most common n-grams in the text. N-grams are contiguous sequences of `n` items from a given sample of text. For example, bigrams are 2-grams (pairs of words), and trigrams are 3-grams (triplets of words). So in other words, n-grams help us identify common phrases or word combinations that might be significant in the context of the text.

You can use the `ngrams` function from the `nltk.util` module to generate n-grams from the list of tokens. You can import it at the beginning of your script:

```python
from nltk.util import ngrams
```

Now, we want to define the `find_ngrams` function. It should take a list of tokens as an argument, an integer `n` to specify the size of the n-grams, and a `top_k` parameter to specify how many of the most common n-grams to return. Here is how you can implement it:

```python
def find_ngrams(tokens, n=2, top_k=10):
```

Inside the function, we can use the `ngrams` function from NLTK to generate n-grams from the list of tokens: `n_grams = ngrams(tokens, n)`. Then, we can use the `Counter` class to count the occurrences of each n-gram: `ngram_counts = Counter(n_grams)`. Finally, we return the most common n-grams using the `most_common()` method: `ngram_counts.most_common(top_k)`. Here is the complete function:

```python
def find_ngrams(tokens, n=2, top_k=10):
    n_grams = ngrams(tokens, n)
    ngram_counts = Counter(n_grams)
    return ngram_counts.most_common(top_k)
```

You can call this function with the `clean_tokens` variable and print the result to see the most common n-grams in the text:

```
common_bigrams = find_ngrams(clean_tokens, n=2, top_k=10)
print(common_bigrams)
```

You can change the `n` parameter to find trigrams or higher-order n-grams. For example, to find the most common trigrams, you can call the function like this:

```
common_trigrams = find_ngrams(clean_tokens, n=3, top_k=10)
print(common_trigrams)
```

Before proceeding to the next step, make sure to comment out or remove the print statement to avoid cluttering your output in the following steps.

6. Finally, we will calculate the TF–IDF scores for the words in the texts. TF–IDF (Term Frequency–Inverse Document Frequency) is a statistical measure used to evaluate how important a word is in a document relative to a collection of documents (the corpus). It highlights words that occur frequently in one text but less often in others — thus identifying terms that are *characteristic* for a given document.

First, we need to import the necessary modules from Gensim. You can add the following imports at the beginning of your script:

```
from gensim import corpora
from gensim.models import TfidfModel
```

Define a function called `find_top_tfidf_words` that takes a list of tokenized texts (`all_tokens`) and returns the top TF–IDF words for each document:

```
def find_top_tfidf_words(all_tokens, top_n=10):
```

Next, we want to create a Gensim dictionary and corpus from the tokenized texts, and then compute the TF–IDF scores. You don't need to worry about the details of how TF–IDF works internally; Gensim handles that for us. But here is how you can implement the function step by step:

Inside the function, we first need to create a Gensim dictionary from the tokenized texts: `dictionary = corpora.Dictionary(all_tokens)` . This dictionary maps each unique word to a unique integer ID. Then we create an empty list called `corpus` to store the bag-of-words representation of each document. We loop through each list of tokens in `all_tokens` , convert it to a bag-of-words format using the dictionary, and append it to the `corpus` list:

```python
corpus = []
for tokens in all_tokens:
    bag_of_words = dictionary.doc2bow(tokens)
    corpus.append(bag_of_words)
```

Next, we create a TF–IDF model using the corpus: `tfidf = TfidfModel(corpus)`. We then apply the TF–IDF model to the corpus to get the TF–IDF representation of each document: `corpus_tfidf = tfidf[corpus]`. The function part should now look like this:

```python
def find_top_tfidf_words(all_tokens, top_n=10):
    dictionary = corpora.Dictionary(all_tokens)
    corpus = []
    for tokens in all_tokens:
        bag_of_words = dictionary.doc2bow(tokens)
        corpus.append(bag_of_words
    tfidf = TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]
```

Now, we want to extract the top TF–IDF words for each document. We can loop through each document in `corpus_tfidf`, sort the terms by their TF–IDF scores in descending order, and select the top `top_n` terms. We can then convert the term IDs back to words using the dictionary and store the results in a list:

```python
top_tfidf_per_doc = [] # Empty list to store top TF–IDF words for each document
for doc in corpus_tfidf: # Loop through each document
    sorted_doc = sorted(doc, key=lambda x: x[1], reverse=True) # Sort terms by TF–IDF score
    top_terms = [(dictionary[term_id], round(score, 4)) for term_id, score in sorted_doc[:top_n]] # Get top n terms and convert IDs to words
    top_tfidf_per_doc.append(top_terms) # Append to the results list
return top_tfidf_per_doc # Return the list of top TF–IDF words for each document
```

Here is the complete `find_top_tfidf_words` function:

```python
def find_top_tfidf_words(all_tokens, top_n=10):
    dictionary = corpora.Dictionary(all_tokens)
    corpus = []
    for tokens in all_tokens:
        bag_of_words = dictionary.doc2bow(tokens)
        corpus.append(bag_of_words)
    tfidf = TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]
    top_tfidf_per_doc = []
    for doc in corpus_tfidf:
        sorted_doc = sorted(doc, key=lambda x: x[1], reverse=True)
        top_terms = [(dictionary[term_id], round(score, 4)) for term_id, score in sorted_doc[:top_n]]
        top_tfidf_per_doc.append(top_terms)
    return top_tfidf_per_doc
```

You can call this function now with the list of all tokenized texts. For this you need to preprocess all texts in the `all_texts` list first. You can do this by using a for loop:

```python
all_tokens = []
for text in all_texts:
    tokens = preprocess_text(text)
    all_tokens.append(tokens)
```

Now you can call the `find_top_tfidf_words` function with the `all_tokens` variable and print the result to see the top TF–IDF words for each document:

```python
top_tfidf_words = find_top_tfidf_words(all_tokens, top_n=10)
print(top_tfidf_words[0])  # Print top TF-IDF words for the first document
```

The output will show you the words with the highest TF–IDF scores in the first document, along with their scores. You can play around with the `top_n` parameter to see more or fewer words and look at other documents by changing the index in `top_tfidf_words`. You can also add custom stopwords to the preprocessing step if you notice that some common words are appearing in the TF–IDF results:

```python
custom_stopwords = ['thee', 'thy', 'thou'] # Add your custom stopwords here
all_tokens = []
for text in all_texts:
    tokens = preprocess_text(text, custom_stopwords=custom_stopwords) # Add custom stopwords variable here
    all_tokens.append(tokens)
```

The TF–IDF value shows how *distinctive* a word is for one text.

- A **high TF–IDF score** means that the word appears often in this text but rarely in others. So it is likely to be more meaningful and characteristic for this document. The highest score is 1.0.
- A **low TF–IDF score** means that the word appears frequently across all texts and is therefore less specific. The lowest score is 0.0.

Example: In a corpus of early modern letters, a word like *merchant* might have a high TF–IDF score in one text written by a trader but not in others — indicating a distinct topic or authorship. Our example corpus is too small to see big differences, but in larger corpora, this method is very useful to identify characteristic terms.

So, what did we learned about the sources?

- The most common words in the text give us an idea of the main themes and topics discussed. For example, if we see words like "witch," "trial," and "accused" frequently, it indicates that the text is likely about witch trials. However, this method simply counts the words without taking any other parameters into account.
- The n-grams help us identify common phrases or word combinations that might be significant in the context of the text. For example, if we find bigrams like "witch trial" or "accused witch," it suggests that these phrases are important in the narrative.
- The TF–IDF scores help us identify words that are particularly relevant to this specific text compared to others in the corpus. Words with high TF–IDF scores are likely to be more meaningful and specific to the content of the text, while common words with low scores may not provide much insight into the unique aspects of the document. This helps us to find special parts of the text that are not just common words but are more specific to the context of the document.

Session 4

Session 4: Data Visualization techniques

In this session, we will explore the fundamentals of data visualization and learn how to use Python tools to represent information graphically.

We will discuss why visualization is important in research, what types of data can be visualized, and how to design effective visualizations that clearly communicate insights.

The practical exercises will focus on Matplotlib (for static visualizations) and DisplaCy (for linguistic visualizations).

Some important concepts

Why Visualizations?

Data visualization transforms complex datasets into clear, interpretable, and engaging visual forms.

In historical and textual research, visualizations help us:

- Reveal patterns, trends, and anomalies in the data.
- Communicate complex findings to others in an accessible way.
- Combine qualitative interpretation with quantitative evidence.
- Support storytelling and argumentation through visual means.

A good visualization does not decorate your data – it helps you think.

What to Visualize

Depending on your research question and data type, you might visualize:

- Text-based data: word frequencies, named entities, or keyword distributions.
- Quantitative historical data: population trends, event counts, or changes over time.
- Relationships and structures: co-occurrences of people, places, or concepts.

The most important step is to match the data type with the right kind of visualization.

For example:

- Frequencies → bar charts or histograms

- Changes over time → line charts
- Relationships → network or dependency graphs
- Linguistic data → entity visualizations (using DisplaCy)

Rules and Best Practices for Visualizations

When designing data visualizations, clarity and honesty are key.

Some essential principles include:

- Clarity first: Remove unnecessary elements and avoid clutter.
- Consistency: Use the same scales, units, and colors where possible.
- Label everything: Axes, titles, and legends make your data interpretable.
- Use color meaningfully: Highlight key insights rather than decorating.
- Provide context: Explain what the viewer is seeing.
- Avoid distortion: Never manipulate scales or proportions to change perception.
- Tell a story: Every chart should communicate a clear message.

Simplicity and clarity are more powerful than complexity.

Libraries for Data Visualization Tasks

Matplotlib

Matplotlib is a widely used Python library for creating static visualizations.

It provides a flexible framework for generating a wide range of plots — including line plots, bar charts, scatter plots, and histograms.

Matplotlib is highly customizable, allowing full control over all visual elements such as colors, fonts, axes, and legends.

Typical use cases in this course:

- Visualizing word frequencies or term distributions.
- Displaying temporal changes in your data.
- Creating comparative bar charts or simple scatter plots.

Example:

```python

import matplotlib.pyplot as plt



plt.bar(words, counts)

plt.title("Word Frequency in Trial Records")

plt.xlabel("Word")

plt.ylabel("Count")

plt.savefig('my_file.png')

```

Use Matplotlib for clear, static visualizations suitable for reports and publications.

DisplaCy

DisplaCy is a web-based visualization tool included in the spaCy library.

It is designed specifically for linguistic data, helping you visualize named entities and syntactic dependencies directly from your text.

Why use DisplaCy:

- It makes linguistic structure visible at a glance.
- You can highlight entities such as names, places, or dates.
- It's excellent for exploring NER (Named Entity Recognition) results from your NLP pipeline.

Example:

```python

from spacy import displacy

doc = nlp("Goody Proctor was accused by Abigail Williams in Salem.")

displacy.serve(doc, style="ent")

```

Use DisplaCy when you want to visualize the results of your text analysis — especially entities and relationships.

Choosing and Evaluating Visualizations

When deciding how to visualize your data, always start with your research question, not with a chart type.

Ask yourself:

1. What story do I want to tell?
2. Who is my audience?
3. What type of data am I working with?
4. Do I want the viewer to explore or to understand?

The best visualization is the one that answers your question most clearly.

Summary

- Visualization is not decoration — it is a form of analysis.
- Every chart should have a clear purpose and message.
- Simplicity, clarity, and consistency are the keys to good visual design.
- Use Matplotlib for static, data-driven plots.
- Use DisplaCy for linguistic and text-structure visualizations.

Exercises

Session 4 - Data Visualization techniques

In this session, we will explore some data visualization techniques using Python. Data visualization is a powerful tool to help you understand and communicate insights from your data.

So far, you have learned the following text analysis techniques:

- Text Preprocessing
- Named Entity Recognition (NER)
- Context Analysis with Word Frequency, N-Grams, and TF–IDF

With these methods, you got interesting insights into the content of the texts. However, the results were mostly presented in your terminal as text output. In this session, we will learn how to visualize the results of text analysis in a more appealing and informative way. Visualizations can help you to better understand the data, identify patterns, and communicate your findings to others.

Exercise 1 - Visualizing NER results with Displacy and Matplotlib

In this exercise, we will visualize the results of Named Entity Recognition (NER) using the Displacy library from SpaCy. Displacy is a powerful tool for visualizing linguistic annotations, including named entities. In session 3, exercise 2 you have already worked with spaCy. We will use the code from that exercise as a starting point. DisplaCy is a part of the SpaCy library. You can read more about DisplaCy in the official documentation . Matplotlib is a widely used library for creating static visualizations in Python. You can read more about Matplotlib in the official documentation .

Remember to install spaCy and download the English model if you haven't done so already:

```
pip install spacy
python -m spacy download en_core_web_md
```

Additionally, we will use Matplotlib to create bar charts that show the frequency of different named entities in the text. You can do so by running:

```
pip install matplotlib
```

1. Open the file `exercise_scripts/exercise_1.py`. In exercise 2 of session 3, we performed Named Entity Recognition (NER) on a text using the SpaCy library. The file `exercise_1.py` already contains the preprocessing steps and the NER function from exercise 3.2, so you can start right there. You need to import the necessary modules from SpaCy. You can do this by adding the following lines after the existing import statements:

```
...

from spacy import displacy
import matplotlib.pyplot as plt
```

These imports will allow you to use Displacy for visualization and Matplotlib for creating bar charts.

2. Now we want to visualize the named entities found in the text using Displacy. Let's create a function called `visualize_entities` that takes a text as input and uses Displacy to visualize the named entities in that text. Your function should look like this:

```
def visualize_entities(text):
```

Now you need to implement the function. Inside the function, you should create a SpaCy document (`doc = nlp(text)`) from the input text and then use DisplaCy to serve a web page that visualizes the named entities. For this, you can use the following code:

```
def visualize_entities(text):
    doc = nlp(text)
    displacy.serve(doc, style="ent", auto_select_port=True, options={"ents":
["ORG", "PERSON"], "colors": {"ORG": "red", "PERSON": "blue"}})
```

We specify that we want to highlight organizations (ORG) in red and persons (PERSON) in blue. Call the function with the text you want to visualize:

```
visualize_entities(our_text)
```

It will open a new tab in your web browser where you can see the visualization. If you want to quit this, just stop the script in your terminal (Ctrl+C).

For our goals, "ORG" is maybe not the most interesting entity type, let's change it to "GPE" (Geopolitical Entity) or "LOC" (Location). Just play around with the entity types to see what you get. You can find a list of some entity types below. You may also recognize that the colors are not very appealing. You can change them to your liking, for example, you could use "GPE": "green" and "PERSON": "orange" or use hex color codes like "#FF5733":

```
displacy.serve(doc, style="ent", auto_select_port=True, options={"ents":
["GPE", "PERSON"], "colors": {"GPE": "green", "PERSON": "orange"}})
```

Label	Description
PERSON	People, including fictional.
NORP	Nationalities or religious or political groups.
FAC	Buildings, airports, highways, bridges, etc.
ORG	Organizations, companies, agencies, institutions, etc.
GPE	Countries, cities, states.
DATE	Absolute or relative dates or periods.
CARDINAL	Numerals that do not fall under another type.

If you are done exploring the Displacy visualization, stop the script in your terminal (Ctrl+C) and delete or comment out (with a #) the call to `visualize_entities(our_text)` to avoid reopening the web server every time you run the script.

3. Now you have visualized the named entities in the text using DisplaCy. But you can also visualize the frequency of the named entities using Matplotlib. We want to create a bar chart that shows the most common named entities in the text. For this, we want to create a function, that plot the entity frequencies:

```
def plot_entity_frequencies(entity_counts):
  labels, values = zip(*entity_counts)
  plt.figure(figsize=(10,5))
  plt.bar(labels, values, color='skyblue')
  plt.title("Most Common Entities")
  plt.xlabel("Entity")
  plt.ylabel("Frequency")
  plt.xticks(rotation=45)
  plt.tight_layout()
  plt.savefig("most_common_entities.png")
```

This function takes a list of entity counts (tuples of entity and frequency) and creates a bar chart using Matplotlib. You can define multiple parameters to customize the appearance of the chart, such as figure size, colors, titles, and labels. If you want, you can play around with Matplotlib and its possibilities.

You can call this function with the most common PERSON entities found in the text:

```
plot_entity_frequencies(most_common_persons)
```

The `most_common_persons` variable is already defined in the code from exercise 2 of session 3. It contains the most common persons found in the text. You can find a png file named `most_common_entities.png` in your working directory after running the script. This file contains the bar chart showing the frequency of the most common PERSON entities in the text.

You can also create a similar chart for GPE or LOC entities by calling the `get_most_common_entities` function with the appropriate entity type and then passing the result to the `plot_entity_frequencies` function. For this, you can use the following code:

```
most_common_locations = get_most_common_entities(our_text, entity_label="GPE")
plot_entity_frequencies(most_common_locations)
```

Play around with the results.

4. Now we want to compare how often different types of entities occur in the text. Create a function called `get_entity_label_counts`. This function should take a text as input and return a dictionary with the counts of each entity label (e.g., PERSON, ORG, GPE, etc.) found in the text:

```
def get_entity_label_counts(text):
    doc = nlp(text)
    label_counts = Counter([ent.label_ for ent in doc.ents]) # Count the occurrences of each entity label
    return label_counts
```

Now we want to plot the results. Create the function `plot_entity_labels`. This function should take the dictionary returned by `get_entity_label_counts` and create a bar chart showing the frequency of each entity label:

```
def plot_entity_labels(label_counts):
  labels = list(label_counts.keys())
  values = list(label_counts.values())
  plt.figure(figsize=(8,6))
  plt.barh(labels, values, color='skyblue')
  plt.title("Distribution of Entity Types")
  plt.xlabel("Frequency")
  plt.tight_layout()
  plt.savefig("entity_label_distribution.png")
```

You can call these functions with the text you want to analyze:

```python
label_counts = get_entity_label_counts(our_text)
plot_entity_labels(label_counts)
```

This will give you a horizontal bar chart showing the distribution of different entity types in the text. You can customize the colors and appearance of the chart as you like. This chart gives you a quick overview of which types of entities are most prevalent in the text.

5. In the NER exercise, you extracted the most common context words around an entity. Now we will visualize them as a bar chart. You can use the `context_templeton` variable from exercise 2 of session 3, which contains the most common context words around the entity "Templeton". Create a function called `plot_context_words` that takes the list of context words and their frequencies and creates a bar chart:

```python
def plot_context_words(context_words, entity_name):
    words, counts = zip(*context_words)
    plt.figure(figsize=(10, 6))
    plt.bar(words, counts, color='#9f66a1ff')
    plt.xlabel('Context Words')
    plt.ylabel('Frequency')
    plt.title(f'Most Common Context Words around "{entity_name}"')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(f'context_words_{entity_name.lower()}.png')
```

You can then call this function with the `context_templeton` variable:

```python
plot_context_words(context_templeton, "Templeton")
```

This will create a bar chart showing the most common context words around the entity "Templeton". Play around with different entities by calling the `get_entity_context` function with other entity names and then passing the results to the `plot_context_words` function.

Now you learned about visualizing NER results using Displacy and Matplotlib. You can explore further by experimenting with different entity types, colors, and texts. Visualizations can greatly enhance your understanding of the data and make it easier to communicate your findings.

Exercises

Session 4 - Data Visualization techniques

In this session, we will explore some data visualization techniques using Python. Data visualization is a powerful tool to help you understand and communicate insights from your data.

So far, you have learned the following text analysis techniques:

- Text Preprocessing
- Named Entity Recognition (NER)
- Context Analysis with Word Frequency, N-Grams, and TF–IDF

With these methods, you got interesting insights into the content of the texts. However, the results were mostly presented in your terminal as text output. In this session, we will learn how to visualize the results of text analysis in a more appealing and informative way. Visualizations can help you to better understand the data, identify patterns, and communicate your findings to others.

Exercise 2 — Visualizing Word Frequencies with Matplotlib

In this exercise, we will visualize the word frequencies from the context analysis (corresponding to exercise 3 of session 3) using the Matplotlib library. The Matplotlib library is a widely used library for creating static visualizations in Python. You can read more about Matplotlib in the official documentation. If you haven't installed Gensim and NLTK in session 3, please do so by running:

```
pip install --upgrade gensim
```

```
pip install --user -U nltk
```

1. First, we need to install the Matplotlib library. You can do this by running the following command in your terminal:

```
pip install matplotlib
```

With Matplotlib installed, we can now proceed to visualize the word frequencies.

2. Now you can open the file `exercise_scripts/exercise_2.py`. In exercise 3 of session 3, we performed a context analysis of a text by finding the most common words using the `word_frequency` function. We will use the results from that exercise to create a bar chart. The file `exercise_2.py` already contains the preprocessing steps and the `word_frequency` function from exercise 3, so you can start right there. There you need to import the necessary modules from Matplotlib. You can do this by adding the following lines after the existing import statements:

```python
...

import matplotlib.pyplot as plt
```

These imports will allow you to use Matplotlib for creating bar charts.

3. Now we want to use the `most_common_words` variable from exercise 3, which contains the most common words and their frequencies. We will create a bar chart to visualize this data. You can do this by adding the following code after the line where `most_common_words` is defined:

```python
# Unzip the most common words into two lists: words and counts
words, counts = zip(*most_common_words)

# Create a bar chart
plt.figure(figsize=(10, 6))
plt.bar(words, counts, color='skyblue')
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Most Common Words in the Text')
plt.xticks(rotation=45)
plt.tight_layout()
plt.savefig('most_common_words.png')  # Save the figure as a PNG file
```

So, what happens here?

- We unzip the `most_common_words` list into two separate lists: `words` and `counts`. This allows us to easily access the words and their corresponding frequencies.
- We create a bar chart using Matplotlib. We set the figure size, bar colors, and labels for the x-axis and y-axis. We also add a title to the chart.
- We rotate the x-axis labels for better readability and use `plt.tight_layout()` to ensure that the layout is adjusted properly.
- Finally, we save the figure as a PNG file using `plt.savefig()`. You should see a file named `most_common_words.png` in your working directory after running the script.

You can play around with the parameters to customize the appearance of the chart. For example, you can change the color of the bars, adjust the figure size, or modify the labels and title. You should also play around with the custom stopwords in line 67 to get better results.

4. We want to use the same code for the most common n-grams. So, we should create a function to avoid code duplication. Define a function called `plot_bar_chart` that takes two lists (words and counts) and a title as input and creates a bar chart:

```
def plot_bar_chart(words, counts, title):
    plt.figure(figsize=(10, 6))
    plt.bar(words, counts, color='skyblue')
    plt.xlabel('Words')
    plt.ylabel('Frequency')
    plt.title(title)
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.savefig(f"{title.replace(' ', '_').lower()}.png")  # Save the figure as a PNG file
```

This code replaces the code you wrote in step 3. The function takes the same parameters as before and creates a bar chart with the given title.

You can then call this function with the `most_common_words` to get the same result as before:

```
# Plot the most common words of the first text
words, counts = zip(*most_common_words)
plot_bar_chart(words, counts, 'Most Common Words in the Text')
```

Plotting the most common bigrams and trigrams is a bit more tricky because the n-grams are tuples of words. You need to convert them to strings before plotting. You can do this by adding the following code after the line where `most_common_trigrams` is defined:

```
# Plot the most common bigrams of the first text
bigram_words, bigram_counts = zip(*most_common_bigrams)
bigram_words = [' '.join(bigram) for bigram in bigram_words] # This converts the tuples to strings with a list comprehension
plot_bar_chart(bigram_words, bigram_counts, 'Most Common Bigrams in the Text')

# Plot the most common trigrams of the first text
trigram_words, trigram_counts = zip(*most_common_trigrams)
trigram_words = [' '.join(trigram) for trigram in trigram_words] # This converts the tuples to strings with a list comprehension
```

```python
plot_bar_chart(trigram_words, trigram_counts, 'Most Common Trigrams in the Text')
```

Now you should see three PNG files in your working directory: `most_common_words_in_the_text.png`, `most_common_bigrams_in_the_text.png`, and `most_common_trigrams_in_the_text.png`. Each file contains a bar chart showing the frequency of the most common words, bigrams, and trigrams in the text, respectively.

If you are done exploring the visualizations of word frequencies, delete or comment out (with a #) the different calls to `plot_bar_chart` to avoid cluttering your working directory with too many files.

5. Now let's continue with the last part: visualizing the TF-IDF results. We will create a bar chart to display the top TF-IDF words in the text. You can use the `plot_bar_chart` function we defined earlier to create the bar chart. Add the following code at the end of the script:

```python
# Plot the top TF-IDF words of the first text
tfidf_words, tfidf_scores = zip(*top_tfidf_words[0])
plot_bar_chart(tfidf_words, tfidf_scores, 'Top TF-IDF Words in the Text')
```

This will create a bar chart showing the top TF-IDF words in the first text and save it as `top_tfidf_words_in_the_text.png` in your working directory. You can customize the appearance of the chart as you like. This chart gives you a quick overview of which words are most important in the text according to the TF-IDF metric. You can also use differeent texts by changing the index in `top_tfidf_words[index]` to visualize the TF-IDF results for other texts in your dataset:

```python
# Plot the top TF-IDF words of the second text
tfidf_words, tfidf_scores = zip(*top_tfidf_words[1])
plot_bar_chart(tfidf_words, tfidf_scores, 'Top TF-IDF Words in the Second Text')
```

6. There are many more ways to visualize text data. Let's try out another one. We will create a word cloud to visualize the most common words in the text. A word cloud is a graphical representation of word frequency, where the size of each word indicates its frequency in the text. It is a popular way to visualize text data because it provides an immediate visual impression of the most important words in a text, but it is not very precise. To create a word cloud, we will use the `wordcloud` library. You can install it by running the following command in your terminal:

```
pip install wordcloud
```

Now you can import the necessary modules from the `wordcloud` library by adding the following lines after the existing import statements:

```python
from wordcloud import WordCloud
```

Next, we will create a function called `generate_word_cloud` that takes a list of tokens as input and generates a word cloud:

```python
def generate_word_cloud(tokens, filename="word_cloud.png"):
    text = ' '.join(tokens)  # Join the tokens into a single string
    wordcloud = WordCloud(width=800, height=400,
background_color='white').generate(text)

    # Display the word cloud using Matplotlib
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis('off')  # Hide the axes
    plt.title('Word Cloud of Most Common Words')
    plt.tight_layout()
    plt.savefig(filename) # Save the figure as a PNG file
```

You can then call this function with the `all_tokens` variable to generate the word cloud. Specify the index of the text you want to visualize. For example, to generate a word cloud for the first text, you can add the following code at the end of the script:

```python
generate_word_cloud(all_tokens[0], filename="first_text_wordcloud.png")
```

This will create a word cloud image of the words in the first text and saves it as `first_text_wordcloud.png` in your working directory. The words that appear more frequently in the text will be displayed in larger font sizes in the word cloud.

You can customize the appearance of the word cloud by changing the parameters of the `WordCloud` class, such as the width, height, and background color. You can also change the filename and the index in `all_tokens[index]` to generate word clouds for other texts in your dataset.

Now you learned about visualizing word frequencies using bar charts and word clouds. You can explore more visualization techniques and libraries to further enhance your text analysis projects. If you understand this exercise, you basically know how to work with different visualizations.

Session 5

Session 5: Working with your own data

In this session, we will focus on applying the techniques and tools we have learned in previous sessions to your own historical text data. For this we will work with the Salem Witch Trial transcripts, which you can find in the `data/` folder. We will go through the process of loading, preprocessing, analyzing, and visualizing your own text data using Python and various libraries such as SpaCy, NLTK, Matplotlib, and others.
You can choose to work with the entire dataset or select specific transcripts that interest you for your research. You can focus on one technique that you found particularly interesting in the previous sessions or combine multiple techniques to gain deeper insights into the text data. Try to formulate a specific research question that you want to answer using the text data. This will help you to focus your analysis and choose the appropriate techniques and tools.

How to start

1. First, make sure you have all the necessary libraries installed in your codespace. We worked with the following libraries in the previous sessions:
   - SpaCy
   - NLTK
   - Gensim
   - Matplotlib
   - WordCloud

You can install any missing libraries using pip.

NLTK:

```
pip install nltk
```

spaCy:

```
pip install -U pip setuptools wheelpip install -U pip setuptools wheel
pip install spacy
python -m spacy download en_core_web_md # choose the model you want to work with
```

gensim:

```
pip install --upgrade gensim
```

matplotlib:

```
pip install matplotlib
```

wordcloud:

```
pip install wordcloud
```

2. Next, you can create a new Python file in the `session_5/` folder. You can name it `exercise.py` or any other name you prefer. To create a new file, you can click on the "New File" button in the file explorer of your codespace and enter the desired name.
3. Now you can start writing your code in the new file. You can use the code snippets and functions from the previous exercises as a starting point. Make sure to import the necessary libraries at the beginning of your script. You can also copy and paste code from the previous exercises if you find it useful for your analysis. There is also a `helpers` folder, which contains all our functions from the previous sessions. You can just copy them into your new file and use them.

And now, have fun exploring your own data!