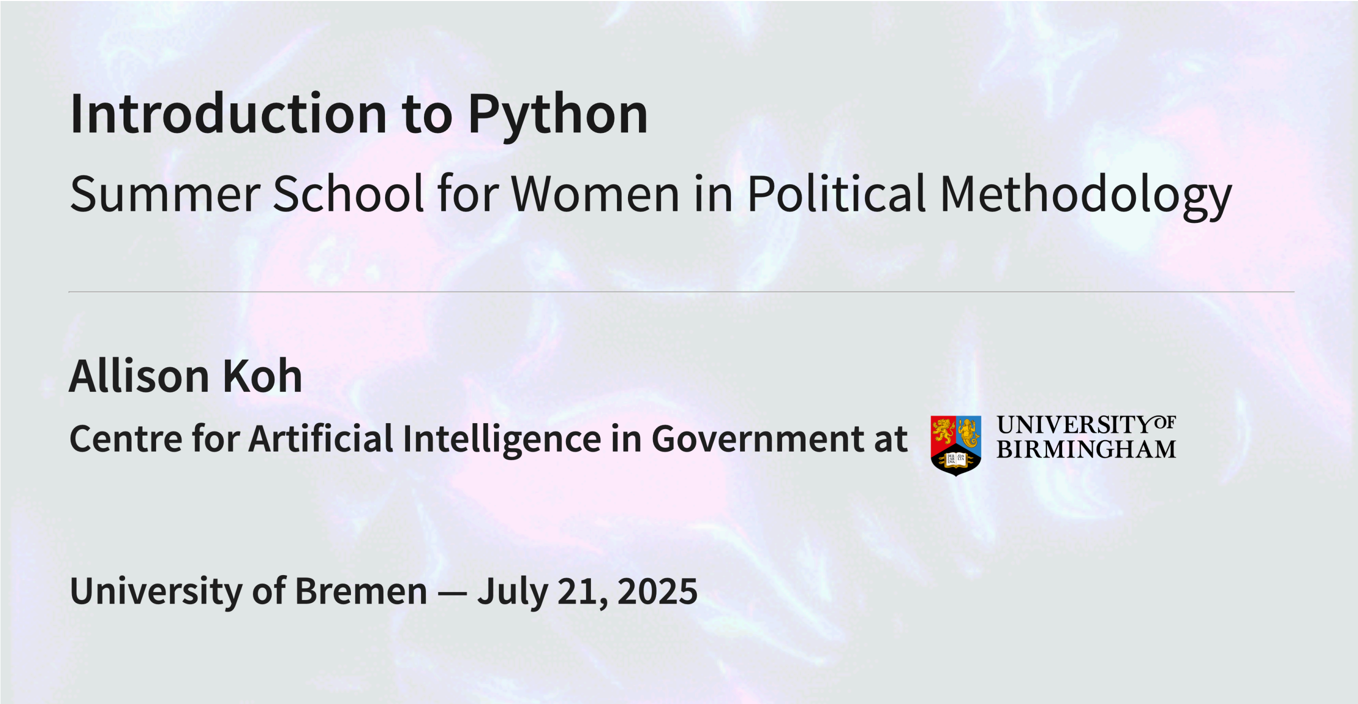
Welcome!



↳ 1 cell hidden

Overview

This afternoon, we will be going over:

- The basics of coding in Python
- Data management with `pandas` 🐼

[ ] ↳ 12 cells hidden

Key Concepts

↳ 6 cells hidden

Python 101

Before we start writing programs in Python, let's first walk through some foundational concepts that explain how Python works under the hood: the core data types and built-in data structures that you'll use all the time.

- Data types

- Assignment
- Strings
- Lists
- Dictionaries

Data types

There are four data types:

- Integers (`int`): numbers without a decimal
- Floating point numbers (`float`): numbers with a decimal
- Booleans (`bool`): `True` or `False` values
- Strings (`str`): Typically a representation of plain text. *However*, anything that is wrapped in quotes (single or double) is treated as a string.

The built-in function to check data types in Python is `type()` . This behaves the same as the `class()` function in `R`.

```
# Integers and floats
x = 5           # int
y = 3.14        # float

# Boolean
z = True        # bool

# String
name = "Allison"  # str
```

And here is how to check the data type of an object:

```
# Check types
type(z)
```

You can convert an object to a different type by calling the type as a function:

```
xx = str(5)
print(xx)
print(type(xx))
```

Assignment

In `R`, we typically use `<-` to assign values to objects. However, in Python, you can only use `=` for assigning values in Python.

Because there is no directional indicator with the equal sign, the values are always on the right hand side while the variable/object will always be on the left-hand side.

```
x = 3
```

```
y = 0.5
```

```
z = True
```

Printing

Use the `print` function if you want an object to be shown:

```
'Hello!'
'Hello world!'
```

```
print('Hello!')
print('Hello world!')
```

Working with strings

Knowing data types is important because you will get different behavior from different operations depending on the type of objects you're working with.

This is especially important for Python because, unlike `R`, you can perform mathematical operations across data types.

Mathematical operations with character strings are referred to as *string operations*. String operations are particularly useful for things like text analysis.

The examples below demonstrate how string operations work.

```
type(1)
```

```
type('1')
```

```
1 + 1
```

When you add two strings using the `+` operator, the two strings are concatenated:

```
'1' + '1'
```

A string multiplied by an integer *n* will repeat the string *n* times:

```
'1' * 5
```

Strings that are placed next to each other are automatically concatenated:

```
'1' '1'
```

```
print("3" + "4")
print("3" * 2)
print(3 + 4)
print(type("3" + "4"))
```

Exercise 1

- Assign the string "5" to a variable called `num_string`.
- Assign the integer `5` to a variable called `num_int`.
- Try printing `num_string + num_int`. What happens?
- Convert one of the variables to fix the error, and write an expression that produces the integer 55.

```
num_string = "5"
num_int = 5
```

```
num_string + num_int
```

```
num_int = str(num_int)
int(num_string + num_int)
```

Lists

Lists are a very common data structure in Python that can hold heterogenous information. They are created with a pair of square brackets. Each element in a list is separated by a comma. e.g.:

```
l = [1, True, "3"]
```

Python is a **0-indexed language**. This means counting starts at 0, and not 1.

If you're accustomed to coding in `R`, this can feel a bit unintuitive at first. This is because R uses 1-based indexing, so the first item in a vector or list is at position 1. In Python, the first item is at position 0, the second at 1, and so on.

This difference is especially important when you're accessing elements in lists or dictionaries, and it's a common stumbling block when switching between the two languages. With practice, though, 0-based indexing becomes second nature!

Subsetting lists

To extract an element from a Python list, you can use one pair of square brackets:

```
l[0]
```

Negative indices

You can select an element from the end of a list using **negative indices**:

```
l[-1]
```

`-1` selects the last element in a list, `-2` selects the penultimate element, and so on...

Slicing

Slicing refers to the extraction of multiple list elements.

To simultaneously select multiple elements you use the colon notation. The first value represents the beginning index, and the second value represents the last index:

```
l = [0, 1, 2, 3, 4, 5]
l[0:3]
```

You can also have an additional third value to specify a step size:

```
l[0:5:2]
```

The indices you pass into the colon notation can be optional.

If you leave the first value blank, Python will slice from the beginning to the second specified index:

```
l[:3]
```

If you leave the second value missing, Python will slice from the first specified index until the end:

```
l[1:]
```

If you leave the first two values blank, and just specify a step size, Python will go through the entire list and return the elements depending on the specified step size:

```
l[::2]
```

Dictionaries

While Python lists are great for storing information, they don't support named elements like R's named lists.

Instead, Python uses **dictionaries** for labeled data, where each item is stored as a key–value pair.

Python dictionaries are created with a pair of curly-brackets. Each key-value pair includes a colon, with the key to the left, and the value to the right of the colon. Multiple key-value pairs are separated by commas:

```
d = {'int_value':1, 'bool_value':True, 'str_value':'three'}
```

You can extract a "value" from the dictionary is by placing the "key" of the dictionary inside the square brackets:

```
d['str_value']
```

Note: Always access a value using its key, not by assuming where it appears in the dictionary.

You can find the number of elements in a list or dictionary by using the `len()` function:

```
print(len(d))
print(len(l))
```

Exercise 2

- Create a list called `characteristics` that contains the following strings: 'first_name', 'last_name', 'age', 'education', and 'employed'.
- Create a dictionary called `person` that assigns values to each of the characteristics.
- Print the full name of your person based on the values assigned in your person dictionary.

```
characteristics = ['first_name', 'last_name', 'age', 'education', 'employed']
person = {'first_name': 'Jane', 'last_name': 'Doe', 'age': 30, 'education': 'PhD'
print(person['first_name'], person['last_name'])
```

Control Flow

Control flow refers to the order in which code is executed. Python provides several control flow statements that allow you to control the flow of your program:

- **if/else statements** allow you to perform different actions based on certain conditions, e.g.:

```
x = 10

if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

- **for loops** allow you to iterate over a sequence of values and perform an action on each value, e.g.:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

- **while loops** allow you to execute a block of code repeatedly while a certain condition is true, e.g.:

```
x = 0

while x < 5:
    print(x)
    x += 1
```

```
0
1
2
3
4
```

- **break and continue statements** allow you to modify the behavior of loops
  - break = exit loop early
  - continue = skip over iterations of a loop

```
for i in range(10):
    if i == 3:
        break
    if i == 1:
        continue
    print(i)
```

```
0
2
```

Exercise 3

Create a list of numbers from 1 to 5. Then:

1. Use a for loop to go through each number
2. If the number is 2, print "Found two!"
3. Otherwise, print "Number:" followed by the number

The output should look like this:

```
Number: 1
Found two!
Number: 3
Number: 4
Number: 5
```

```
numbers = [1, 2, 3, 4, 5]

for num in numbers:
    if num == 2:
        print("Found two!")
    else:
        print("Number:", num)
```

```
Number: 1
Found two!
Number: 3
Number: 4
Number: 5
```

Functions, methods, and attributes

As previously discussed, functions work similarly in both `R` and Python. However, while `R` relies heavily on functions, Python relies on functions, methods, *and* attributes.

Functions work similarly in both `R` and Python. This is what the Python syntax looks like for functions:

```
def greet(name):
    print("Hello,", name)
```

While `R` relies heavily on functions, Python relies on functions, methods, *and* attributes.

(*Note:* attributes will be demonstrated later, since the data structures we have seen thus far do not have any meaningful attributes for us to play around with)

Appending lists

You can add elements to a list using the `append()` method:

```
l = [0, 1, 2, 3, 4, 5]
l.append(6)
l
```

Note the use of the dot notation to call the `append` method from the `list` object.

Updating dictionaries

For dictionaries, you can call the `update()` method:

```
d = {'lol':'laugh out loud', 'idk':'i dont know', 'fml':'f my life'}
d.update({'dm': 'dont mind', 'afaik': 'as far as i know'},)
d
```

If the key already exists, the dictionary will be updated with the new value. Otherwise, new keys will be added to the dictionary (as above).

```
d.update({'dm': 'direct message'})
d
```

Exercise 4

- What happens when you try to run the following code chunk? Why?
- Edit the code to address this error.

```
d.append({'rofl': 'rolling on the floor laughing'})
```

Answer: We see an error message beacuse dictionary objects do not have an 'append' attribute. If we want to augment lists, we use the 'append' attribute. If we want to augment dictionaries, we use the 'update' attribute.

```
# Write the correct code here
d.update({'rofl': 'rolling on the floor laughing'})

# Check that it works
d
```

Libraries

Libraries in Python are similar to packages in `R`: they extend the base language with specialized tools and data structures.

Many of the objects that are widely used in data science are not built into Python by default. To use them, you'll need to import external libraries.

Two libraries you'll likely use in nearly every data analysis script are:

- `numpy`: Provides support for **numeric arrays and matrix operations**
- `pandas`: Offers the `DataFrame` object for working with tabular data (similar to `dplyr` in `R`)

When you load a package in `R`, you can use any of the functions from that package directly. However, things work a little differently with Python.

To load in a Python library, you use the `import` keyword, e.g.:

```
import numpy
```

It is also useful to have information on the versions of libraries you use in your code; this is helpful information for both replicability and troubleshooting:

```
numpy.__version__
```

Once you import the library, you have to access functions from the library using the dot notation: `library_name.function_name`, e.g.:

```
numpy.array([1, 2, 3])
```

Typing the full library name each time can get tedious. Since you *have* to use library names to call the functions attached to them, Python allows you to create a shortcut called an *alias*.

It's common practice to import NumPy as `np` and Pandas as `pd`:

```
import numpy as np
import pandas as pd
```

Now, instead of writing `numpy.array`, you can write:

```
np.array([1, 2, 3])
```

Data management with `pandas` 🐼

As previously mentioned, we'll be using the **Parties' Immigration and Integration Positions Dataset (PImPo)** for our applied examples. The dataset, which is an expansion of the data resources offered by the Manifesto Research on Political Representation (MARPOR), includes information on the following variables:

- `country`: MARPOR country id
- `party`: MARPOR party id
- `date`: election date
- `totals`: total quasi-sentences (QSs) coded
- `totals_immi`: total QSs on immigration
- `totals_inti`: total QSs on integration
- `saliency`: the proportion of QSs related to immigration and integration relative to the total QSs coded
- `saliency_immi`: proportion of immigration-related QSs relative to total QSs coded
- `saliency_inti`: proportion of integration-related QSs relative to total QSs coded
- `immi_pos`: a score that indicates how positively or negatively a party talks about immigration
- `immi_pos_saliency`: the percentage of directional (non-neutral stance) QSs about immigration
- `inti_pos`: a score that indicates how positively or negatively a party talks about integration
- `inti_pos_saliency`: the percentage of directional (non-neutral stance) QSs about integration

Loading datasets

Now that we've got everything we need, let's play around with data! First, let's import the `pandas` library under the alias `pd`:

```
import pandas as pd
```

Now let's load in our data using the `pandas.read_csv` function:

```
df = pd.read_csv("PImPo_party.csv")
```

Note: See the above section on [data](#) if you need to first upload the dataset from your local files.

Data frame at a glance

Let's inspect the data.

```
df.head()
```

Here you'll see that there are three components to a `Pandas` data frame:

- Column names on the top
- Index on the left
- Body of the data frame

Each of these components are *attributes* that can be accessed independently of one another, e.g.:

```
df.columns
```

Note that `()` is not necessary for this call since `columns` is not a function or method.

Calling a data frames *index* may be helpful when working with time series data; you can replace the original index with date-time information. This is how you access it:

```
df.index
```

The `shape` attributes store the dimensions of the data frame.

```
df.shape
```

The `values` attribute returns the body of the data frame:

```
df.values
```

If you don't have too many columns, `info()` is useful for summarizing what's in the data.

```
df.info()
```

Exercise 5

What do we now know about the data from this information, that we didn't know from looking at the `index`, `shape`, and `values` attributes?

Answer: Here's what we can learn a lot after applying the `.info()` method that we didn't already know from applying the other functions:

- Data types: 4 integer columns, 9 float columns
- Missing data
    - The `totals_immi`, `immi_pos`, `immi_pos_saliency`, `totals_inti`, `inti_pos`, and `inti_pos_saliency` variables have missing values.
    - The immigration-related and integration-related variables have the same number of missing values (200 non-null for immigration-related variables, 208 non-null for integration-related variables).
- Memory usage is relatively small: 24.7KB

Subsetting data

Now that we've had a look at the data, let's do something with it!

Selecting a single column

If you want to extract a single column from a pandas DataFrame, you can use the square bracket notation:

```
df['country']
```

These numbers aren't the most useful for us; even with a key, it would be easier/more readable for the country names to be included in the data. We'll get to that in a bit.

We can take this column and save it to its own variable:

```
country_df = df['country']
```

Then we can preview only the first few rows using the `.head()` method:

```
country_df.head()
```

Let's check out the type of `country_df`:

```
type(country_df)
```

This tells us that a single column of a DataFrame is a Series, *not* a DataFrame. While they are very similar and interrelated, this also means that each of these object types have their own set of methods and attributes.

Selecting multiple columns

Let's say you want to select multiple columns from the dataset—specifically, information about the country, political party, and the total number of quasi-sentences coded.

Here's how you can do that and store it in a list:

```
subset = df[['country', 'party', 'totals']]
subset.head()
```

The double square brackets indicate that you are both taking the subset of the data frame and storing it as a list.

Filtering rows with `.loc[]` and `.iloc[]`

In pandas, there are two ways to access specific rows from a DataFrame: `.loc[]` and `.iloc[]`.

`.loc[]` (location) retrieves rows based on the row label, which is what we saw in our initial glance of the data. It's not about the row's numeric position, but what the index is *named*:

```
df.loc[2]
```

Note that, if you end up combining datasets, you may end up with two rows that have the same index number if the index labels correspond to the respective original datasets. So `.loc[]` can be a bit tricky in that way.

On the other hand, `.iloc[]` (index location) identifies rows based on their position. Recall that Python is 0-indexed. This means that, for example, `iloc[2]` will always return the third row, which is indexed at position 2.

```
df.iloc[2]
```

If you want to access multiple rows, you can put that set in a python list (like we did with columns), e.g.:

```
df.iloc[[5, 2]]
```

Extracting multiple rows and columns based on location

We use `.loc[]` to subset both rows and columns in a `DataFrame`. The syntax is similar to how subsetting works in `R`: **rows go on the left, columns go on the right, separated by a comma** inside the brackets.

```
df.iloc[0:5, 0:4]
```

As we reviewed with slicing elements of a list, we can use shortcuts to indicate ranges of rows and/or columns:

```
df.iloc[:5, :4]
```

Filtering rows with boolean subsetting

Sometimes, instead of selecting rows by their position or label, it makes more sense to filter them based on the values in one or more columns. This is known as *boolean subsetting*.

In pandas, you can create a condition that--under the hood--returns `True` or `False` for each row, and use that to filter the DataFrame.

Let's say we are only interested in looking at parties from Germany (country code `41`), and their positions on immigration (`immi_pos`). This is how we would subset the data to get this information:

```
df.loc[df['country'] == 41, ['date', 'country', 'party', 'immi_pos']]
```

Filtering rows with multiple boolean subsetting

Let's say we now want to filter based on two criteria:

- Germany (`country` = 41)
- anti-immigrant positions (`immi_pos` < 0)

To do this, we wrap each condition in its own set of parentheses and connect them using a logical operator (`&` for “and”, `|` for “or”):

```
df.loc[(df['country'] == 41) & (df['immi_pos'] < 0), ['date', 'country', 'party',
```

Exercise 6a

Run the two code chunks below. What do you notice about the similarities and differences in their outputs? Why are they behaving this way?

Hint: Have another glance at the `.head()` of the data frame to inspect how the rows are labeled.

```
df.loc[3]
```

```
df.iloc[3]
```

```
df.head()
```

Answer: The outputs are the same because the row numbers and the index labels are the same. In other words, the row labels are accessed by `.loc[]` match the integer positions accessed by `.iloc[]`.

Exercise 6b

Use boolean filtering to create a data frame with information on the immigration and integration positions of political parties from the U.S. and Canada.

Hint: Have a look at the codebook: https://manifesto-project.wzb.eu/down/datasets/pimpo/PImPo_codebook.pdf#page=9.60

```
df.loc[(df['country'] == 61) | (df['country'] == 62), ['country', 'date', 'party'
```

Tidy data management

Let's transform this data for analysis. We will select the variables we are interested in, reshape the data, and compute grouped summaries. If you have used `dplyr` or `tidyr` in R, some of the Python syntax will feel similar.

For your reference, here are some of the more common data wrangling tasks and their implementation in Python/ R :

Task	Description	Python (pandas)
Recoding variables	Map values using a dictionary/lookup	`df['column'].map(dictionary)`

Task	Description	Python (pandas)
Selecting columns	Keep only specified columns	`df[['col1', 'col2', 'col3']]`
Reshaping wide to long	Convert wide format to long format	`pd.melt()`
Dropping missing values	Remove rows with missing values	`df.dropna()`
Converting data types	Change column data types	`df['col'].astype()`
Date parsing	Convert strings/integers to datetime	`pd.to_datetime()`
Extracting date parts	Extract year, month, etc. from dates	`df['date'].dt.year`
Grouping data	Group data by specified variables	`df.groupby(['col1', 'col2'])`
Summarizing grouped data	Calculate summary statistics by group	`.groupby().mean()`, `.sum()`, `.count(`
Resetting index	Convert grouped result back to regular DataFrame	`df.reset_index()`

Let's say we want to **compare the average immigration and integration positions of political parties across countries and over time**. With this goal in mind, we need to:

1. Make the dataset more interpretable by mapping country codes to meaningful names.
2. Only keep the columns we care about.
3. Reshape the data from wide to long.
4. Drop missing values.
5. Convert the date variable to `datetime` format
6. Group and summarize the data by country and year.

Mutating/recoding variables

Based on the codebook, we can create a dictionary that maps `country` codes onto country names:

```
country_labels = {
    13: 'Denmark',
    14: 'Finland',
    22: 'Netherlands',
    33: 'Spain',
    41: 'Germany',
    42: 'Austria',
    43: 'Switzerland',
    53: 'Ireland',
    61: 'USA',
    62: 'Canada',
    63: 'Australia',
    64: 'New Zealand'
}
```

We can then `map` this dictionary onto the dataset:

```
df['country'] = df['country'].map(country_labels)
```

Great, let's check to see that this worked:

```
df.head()
```

Select columns of interest

What are the relevant variables that we need to select for this exercise?

Let's have a look at what the variables are:

```
df.columns
```

Based on this information, we see that we only need 5 variables:

- `country`
- `party`
- `date`
- `immi_pos`
- `inti_pos`

```
df_subset = df[['country', 'party', 'date', 'immi_pos', 'inti_pos']]
```

Great, let's have another look at the data:

```
df_subset.head()
```

Reshaping the data from wide to long

Right now, immigration and integration positions are stored in separate columns — this means our data is in wide format. To make it easier to summarize, reshape, or visualize, we'll use `pd.melt()` to convert the dataset into long format:

```
df_long = pd.melt(
    df_subset,
    id_vars=['country', 'party', 'date'],
    var_name='position_type',
    value_name='position_score'
)
```

Great, let's have a look at the resulting data frame:

```
df_long
```

Dropping rows with missing values

We see that there are some NaN values under the `position_score` variable. This is going to impede on our goal of grouping and summarizing information from this variable.

Let's deal with these missing values. To do this, we drop any rows where `position_value` is missing using Panda's `.dropna` method:

```
df_complete = df_long.dropna(subset=['position_score'])
```

Great, let's check to see if this removed anything. We can do this using a combination of two methods:

- `.isna()` : returns `True` for missing values, `False` otherwise
- `.any()` : checks for the presence of something; returns `True` if it is found and `False` otherwise

Let's look at these one by one. This is what happens if you just use `.isna()` :

```
df_long['position_score'].isna()
```

It does the job, but we can present this information more cleanly. This is where `.any()` comes in:

```
df_long['position_score'].isna().any()
```

Now let's check for any missing values in `df_complete` :

```
df_complete['position_score'].isna().any()
```

Working with dates 📆

Dates in raw datasets are often stored as integers or strings. However, to work with them meaningfully (e.g. group by year, extract months, compare time periods), we need to convert them to a `datetime` format.

If you've used `lubridate` in `R`, this is similar to using functions like `ymd()` or `as_date()` to unlock date-aware functionality.

Pandas has a method called `to_datetime` that converts a variable into `datetime` format.

To get an idea of how this works, let's first inspect the type of variable we're starting out with. Let's try applyin the `type()` function from before:

```
type(df_complete['date'])
```

Did this give us the information we wanted? **Nope!** We instead got the type of the entire column, which is a `Series` as previously discussed.

In `pandas`, when we are interested in inspecting a variable's type, we call the `.dtype` method:

```
df_complete['date'].dtype
```

Okay, so we're starting out with an integer. We're going to need to convert this to a string variable if we want to then parse it to `datetime` format:

```
df_complete['date'] = df_complete['date'].astype(str)
```

Let's create a new variable `year_month`, which is a `datetime` object based on the existing `date` variable:

```
df_complete['year_month'] = pd.to_datetime(df_complete['date'], format='%Y%m')
```

We can use the `datetime` Python library to further parse our `year_month` variable:

```
import datetime as dt
```

```
df_complete['year'] = df_complete['year_month'].dt.year
df_complete['month'] = df_complete['year_month'].dt.month
```

```
df_complete.info()
```

Grouping and summarizing the data

Now that we have prepared our data, let's get to computing the average party positions regarding immigration and integration, disaggregated by country-years.

As with before, the syntax we use to group and summarize pandas data frames should seem familiar to similar functions if you are used to working with `dplyr`'s `group_by()` and `summarize()`.

The `.groupby()` method groups the data based on specified variable(s):

```
df_complete.groupby(['country', 'year'])
```

However, nothing happens with the data until you combine `.groupby()` with an aggregation method such as:

- `.mean()`
- `.count()`
- `.sum()`

```
df_avg_scores = df_complete.groupby(['country', 'year', 'position_type'])['positi
df_avg_scores.head()
```

Great, looks like we're getting somewhere! Let's just check one thing:

```
type(df_avg_scores)
```

Uh oh, that's not a data frame! Sometimes, when you're subsetting data frames to create new data frames, you should check if Python has recognized the new data frame as its own data frame.

We can remedy this with the `.reset_index()` method:

```
df_final = df_avg_scores.reset_index()
```

Nearly there; now that the calculations have been made, this data frame would probably be a little easier to read/more interpretable in its wide format. That said, time for another exercise!

Exercise 7

1. Reshape the data back to wide format
   - Create a new DataFrame called `df_wide` where immigration and integration scores are stored in separate columns.
   - *Hint:* Use the `.pivot_table()` method.
2. Inspect divergences between integration and immigration positions at the country level
   - Using your new `df_wide` DataFrame, create a new variable `gap` which computes the difference between `immi` and `inti` scores.
   - Which country has the largest *positive* divergence between their positions on immigration and integration?
   - Which country has the largest *negative* divergence between their positions on immigration and integration?

Hint: Use the pandas documentation if you are unsure how to apply the suggested methods.

```
df_wide = df_final.pivot_table(index=['country', 'year'], columns='position_type'

df_wide


df_wide['gap'] = df_wide['immi'] - df_wide['inti']


df_wide.head()


# Find the country/year with the largest positive gap (immi > inti)
largest_positive_gap = df_wide['gap'].___()
country_largest_positive_gap = df_wide[df_wide['gap'] == largest_positive_gap]

# Find the country/year with the largest negative gap (inti > immi)
largest_negative_gap = df_wide['gap'].___()
country_largest_negative_gap = df_wide[df_wide['gap'] == largest_negative_gap]

print("Country/Year with the largest positive gap:")
print(country_largest_positive_gap)

print("\nCountry/Year with the largest negative gap:")
print(country_largest_negative_gap)
```

Bonus Exercise (if time permits)

Write a script that wrangles data in Python based on an `R` script and/or dataset from your research.

References

Lehmann, P., & Zobel, M. (2018). Positions and saliency of immigration in party manifestos: A novel dataset using crowd coding. *European Journal of Political Research*, 57(4), 1056-1083. https://doi.org/10.1111/1475-6765.12295