Introduction to Python programming

Léo Picard

Spring semester, 2024

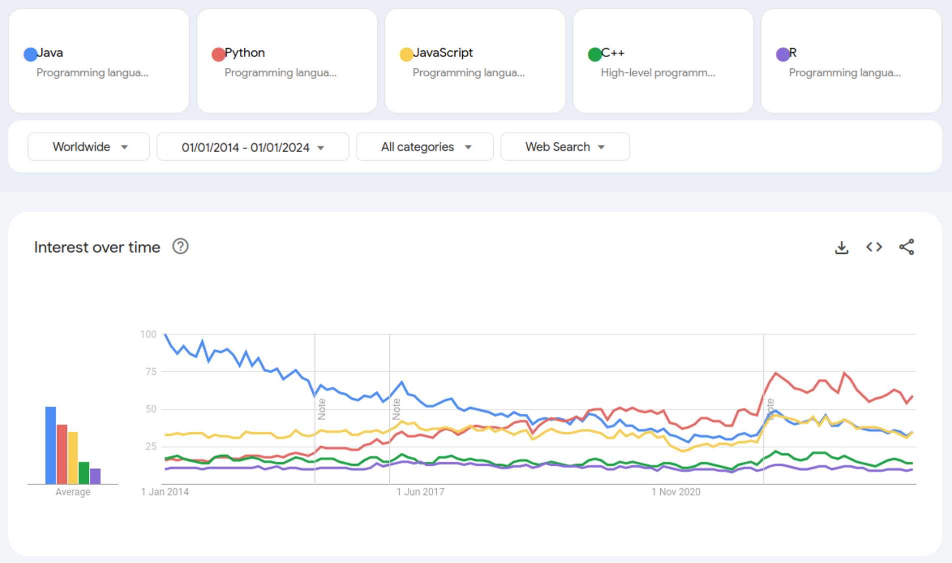
Course material available on:
https://github.com/leops95/intro_to_python

leo.picard@unibas.ch

What's Python?

- General-purpose programming language
- Free and open source
- Elegant and user-friendly syntax
- Many useful libraries (Pandas, NumPy, Matplotlib, OpenCV, NLTK, statsmodels, Scikit-learn, PyTorch...)

One of the most popular programming languages



Learning curve

- The learning curve is hard at first
- It gets easier with experience:
  - knowing the syntax and the tools
  - your past projects can still help you when you're stuck
- No one knows everything by heart
- My goal is to show you the basics and help you to **become independent**

Objectives

1. **Set-up**: Install and use Python
2. **Python essentials**: The syntax, data types and basic operators
3. **Scientific computing**: Load datasets and work with them, plot data
4. **Asking for help**: Becoming independent online

Before we start

- This course is for **you**, I'm adapting to your needs
- Tell us a bit about yourself !
  - Have you ever used Python ?
  - Why would you like to learn Python ?
  - Do you have any other programming experience ?

Sections

Set-up

Installation

- To install the "core Python package" you can go to `https://www.python.org/`
- As we want to use Python for scientific programming, you only have to install "Anaconda": `https://www.anaconda.com/`

→ Anaconda is a free distribution for Python which provides the core Python package and the most popular scientific libraries

- We write and compile code ("scripts") in files with the following extension: `filename.py`

Setting up your environment

Definition

The **IDE (Integrated Development Environment)** is the software we're using to run python scripts

Different IDEs for different needs:

- Very light: problem sets, step-by-step tutorials (ex: Jupyter Notebook, Google Colab...)
- Intermediate: built-in data viewer (ex: Spyder)
- Heavy but efficient: for big projects and software engineering (ex: VS Code...)

Setting up your environment

- We will use the Spyder IDE which comes with Anaconda
- Load it either from the Anaconda navigator or using the terminal
- Spyder is split into different "panes" which are sections providing us with information or access to certain features. The most important are:
  - The editor
  - The console
  - The variable explorer and plots
- You can add, move or remove panes (see "View" → "Panes")

Python essentials

Basics

- Using hashtags (#), we take notes ("comments") directly into the code
- Enclosing lines within quotation marks (""") makes multi-line comments
- To display something on the console, we use the `print()` function
- I use the symbol > at the start of a line to show the result on the console

```
# the command below is likely going to be the
# first thing you try in any programming language
print("Hello world!")

> "Hello world!"
```

Note: Most IDEs have a color scheme to distinguish different elements of code

Variables

- **Variables** store data in our programs
- Using the assignment operator "=", we give them names and values
- Variables can take different data types: numbers, text, they could be binary, complex, numbers, contain a tuple, a list, even a dictionary !
- the variable explorer shows you the type of all variables you have created

```python
# assign values to variables
number_1 = 15
my_name = "Leo"
num_list = [2, 5]
```

Multiple assignment

You can assign multiple values to do different variables in one line

```
# assign values to variables
number_1 = 15
my_name = "Leo"
num_list = [2, 5]

# delete them
del number_1, my_name, num_list

# assign them again all at once
number_1, my_name, num_list = 15, "Leo", [2,5]
```

Numbers

- There are two different types of data representing numbers
  - **Integers** (`int`): whole numbers (0, 1, 2, 5001, -9999)
  - **Floats** (`float`): numbers with decimals (1.1, 2.64, 6.666666...)
- Python may dynamically change variable types if values are affected

```
number_1, number_2 = 1.99, 15
type(number_2)

> <class 'int'>

number_2 = number_1 + number_2
type(number_2)

> <class 'float'>
```

Strings

- A **string** (`str`) is a series of characters
- In Python anything inside single or double quotes are strings `"My name is..."` or `'Python is fun!'`
- We can also use both within the same string, **but only when they are nested** `'He said, "I love my dog."'`
- Using **F-strings**, we can enter any variable value within a string

```
name, birth = "Léo", 1995
sent = f"Hi ! My name is {name} and I'm {2023-birth} years old."

print(sent)

> "Hi ! My name is Léo and I'm 28 years old."
```

Booleans

- A **boolean** (`bool`) is a data type that has two possible values (`True` or `False`)
- They are often used to keep track of conditions
- But usually we get them from doing logical comparisons (ex: `2 == 3` $\rightarrow$ `False`)

```
boolname = False
print(boolname)

> False

boolname = (5**2 == 25)
print(boolname)

> True
```

Lists

- A **list** (`list`) is a sequence of **elements** (or items) in a particular order
- You can modify an element of a list by accessing it

```
listname = [1,4,5,8]

print(listname[2])

> 5

listname[2] = 7

print(listname)

> [1,4,7,8]
```

Lists

- Lists are mutable, which means we can change the order (**index**) of elements
- The following table shows the most important list methods

Method	Description
`listname.append(i)`	Add an item `i` at the end of the list
`listname.insert(x,i)`	Insert an item `i` at the position `x`
`listname.pop(x)`	Remove item at position `x` and return it
`listname.copy(x)`	Return a copy of the list
`listname.sort()`	Sort all the items in the list (increasing by default)

Important

The index position in Python starts at 0, not 1 (sorry Matlab users!)

Lists

Example	Outcome
`a = [1,2]; a.append(3)`	`> a = [1,2,3]`
`a = [1,2]; a.insert(1,3)`	`> a = [1,3,2]`
`a = [1,2,3]; popped = a.pop(1)`	`> a = [1,3]; popped = 2`
`a = [1,2]; b = a.copy()`	`> a = [1,2]; b = [1,2]`
`a = [4,1,5,3]; b = a.copy();` `a.sort(); b.sort(reverse = True)`	`> a = [1, 3, 4, 5]; b = [5, 4, 3, 1]`

Slicing lists

To select some elements in a list, we **slice** it using: `listname[a:b]` (**b is excluded**)

```
colors = ["red", "green", "blue", "yellow"]

print(colors[1:3]) # elements 1 and 2

> ['green', 'blue']

print(colors[1:]) # last three elements

> ['green', 'blue', 'yellow']

print(colors[-1:]) # last element

> ['yellow']
```

Dictionaries

- **Dictionaries** (`dict`) are used to store data in pairs (key + value)
- They do not allow duplicates, elements can be retrieved by their key
- Assigning values to a new key creates a new element

```
dictname = {"BS": "Basel Stadt", "GE": "Geneva", "TI": "Ticino"}
print(dictname["BS"])

> "Basel Stadt"

dictname["ZH"] = "Zurich"
print(dictname)

> {'BS': 'Basel Stadt', 'GE': 'Geneva', 'TI': 'Ticino', 'ZH': 'Zurich'}
```

Dictionaries

- Dictionaries (and lists), can be **nested**

→ they can contain another dictionary, or data type

```
dictname = {"owners": ("Antonia", "Elda"),
            "pets": {"dogs": ("Charlie", "Razmotte", "Nemo"),
                     "cats": ("Zazie", "Peps", "Zélie")}}

print(dictname["pets"]["dogs"])

> ('Charlie', 'Razmotte', 'Nemo')
```

Arithmetic Operators

	Operator	Example
Addition	`+`	`10 + 5 = 15`
Subtraction	`-`	`30 - 20 = 10`
Multiplication	`*`	`2 * 5 = 10`
Division	`/`	`6 / 2 = 3.0`
Modulus	`%`	`10 % 4 = 2`
Exponent	`**`	`2 ** 3 = 8`
Floor Division	`//`	`9 // 4 = 2`

Note: `^` is the bitwise operator "xor" (exclusive or)!

Comparison Operators

Operator	Description	Example
`==`	equal	`4 == 3` → `False`
`!=`	not equal	`4 != 3` → `True`
`>`	greater than	`6 > 10` → `False`
`<`	less than	`2 < 5` → `True`
`>=`	greater or equal	`8 >= 3` → `False`
`<=`	less than or equal	`5 <= 5` → `True`

Boolean Operations

Suppose `x = True` and `y = False`

Operation	Result
`x or y` → `True`	if `x` is false, then `y`, else `x`
`x and y` → `False`	if `x` is false, then `x`, else `y`
`not x` → `False`	if `x` is false, then `True`, else `False`

Conditions

If statements (`if`) execute a piece of code only if a **condition** is satisfied (`True`)

```
x, y = 5, 10

if y < x:
    print("y smaller than x")
else:
    print("y greater than x")

> "y greater than x"
```

- the `else` block runs only if the condition is not satisfied (`False`)
- For more than two conditions, you can insert `elif` ("else if") before `else`
- Be careful of the indentation !

For loops

- Often, we want to perform the same task repeatedly or with each item in a list
- **For statements** (`for`) iterate over items, in the index order
- Iterating does not make a copy of the sequence

```
numbers = [4,34,2]

for number in numbers:
    print(number + 1)

> 5
> 35
> 3
```

List comprehension

To iterate over all elements of a list, using brackets as **list comprehension** are more efficient

```
listname = [1, 2, 3, 4, 5, 6]

listname = [x*x for x in listname]
print(listname)

> [1, 4, 9, 16, 25, 36]

# we can even add conditions
listname = [x for x in listname if x%2 == 0]
print(listname)

> [4, 16, 36]
```

How many loops ?

- The `range(a, b)` function generates arithmetic progressions
- As with lists, the last element (`b`) is excluded
- It is commonly used to loop a specific number of time in `for` loops
- You need to name the current item (below, `i`), if you want to use it inside the loop

```
for i in range(1, 4):
        print("Loop number", i)

> Loop number 1
> Loop number 2
> Loop number 3
```

How many loops ?

The `len()` function gives you the length of a list

```
floats = [1.2, 2.343, 0.44]

for i in range(len(floats)):
        print(i, floats[i])

> 0 1.2
> 1 2.343
> 2 0.44

# another example with list comprehension
list_loop = [2*i for i in range(5)]
list_loop

> [0, 2, 4, 6, 8]
```

While loops

- **While statements** (`while`) execute a task repeatedly *while* a condition is true
- You can also stop the loop using `break`

```
i = 1
while i < 10:
        print(i)
        if i == 4:
                break
        i += 1 # equivalent to i = i + 1

> 1
> 2
> 3
> 4
```

Functions

Definition

A function is a block of code that is written to do a specific task, upon calling its name

- It saves time as we don't have to repeat the same code
- By using the keyword `def` we tell python that we are **defining** a function
- It is followed by the function name and a list of parameters in parentheses
- After the function is defined, we **call** it with the required parameters

Functions

An example using the Fibonacci series:

```
def fib(n):
        """
        Print a Fibonacci series up to n
        """
        a, b = 0, 1
        while a < n:
                print(a, end = ' ')
                a, b = b, a + b

fib(10)
```

> 0 1 1 2 3 5 8

Functions

Functions can **return** an output, which will be stored in a variable (if assigned)

```
def squared(array):
        """find the square of each element in a vector"""
        output = []
        for elem in array:
                elem_squared = elem**2
                output.append(elem_squared)
        return output

n = [2, 5, 10]
n_squared = squared(n)

print(n_squared)

> [4, 25, 100]
```

Lambda expressions

Named functions (with `def`) are time-wise inefficient for simple operations

Instead, we can use **lambda expressions**: `(lambda x: operation)(value)`

```
# Named function
def simple_operation(x):
        x_new = x**2-1
        return x_new

n_new = simple_operation(10)
print(n_new)
> 99

# Same with lambda expression
(lambda x: x**2-1)(10)
> 99
```

Now it's your turn !

Some exercises to practice:

1) Create two variables, then swap their values
2) Create a list containing the numbers 0 to 9, then invert it (9 to 0)
3) Write a function that returns the square of all odds or even numbers between 0 and 20

The file `solutions.py` contains the answers

Scientific computing

Paths

Definition

Your computer stores files in directories, which can be accessed using **paths**. It comes in different formats depending on your operating system

Let's take the Desktop:

- For Windows: `C:\Users\picard0001\Desktop`
- For MacOS: `/Users/picard0001/Desktop`
- For Linux Ubuntu: `/home/picard0001/Desktop`

Simply replace "picard0001" by your own session user name

Note: `~\Desktop` is also valid

Paths

- The Python console is always looking at one directory
- You can show which one using the command `pwd` ("print working directory")
- Paths can be absolute and relative.
  - Absolute paths refer to the entire path to your destination
  - Relative paths refer to paths *relative* to the current directory
- Changing the directory is easy: either enter a new (absolute) path or go up/down the path tree with the (relative) path

Packages

Definition

Packages are a collection of modules (Python files) that we **import** into our code. They contain functions that serve a purpose, and are ready to be used.

- First, search a package name on the internet, find the command to install it
  - `https://pypi.org/`
  - `https://anaconda.org/conda-forge/`
- Then, paste the command on the terminal with a package manager:
  - **Pip**: the default one (`pip install pandas`)
  - **Conda**: the Anaconda version (`conda install -c conda-forge pandas`)

Packages

Installing new packages can be tedious, because:

- you need to use the terminal (with Bash commands) to install them
- they come in different versions
- they need to be stored in a specific location (the "$PATH") where Python will look for them
- they can enter in conflict with other packages

No need to worry about the $PATH with Anaconda. Otherwise, here are nice tutorials on using Bash commands **[Click here]** and managing $PATH **[Click here]**

Packages

Finally, we import a package into our code using an `import` statement

```
import numpy

# draw two random values (normally distributed)
print(numpy.random.randn(2))

> array([-1.0856306 ,  0.99734545])
```

- Subpackages only contain some functions
- We call them by using a point after the packge name (e.g. "numpy.random")
- Calling `import numpy.random` instead of `import numpy` saves a lot of memory !

Packages

- The keyword **as** names the package differently
- The keyword **from** calls only specific subpackages or functions

```
import numpy as np
from numpy import cos, pi

print(np.sin(np.pi)) # "np" is way shorter than "numpy"

> 1.2246467991473532e-16

print(cos(pi)) # with "from" we can even omit "np." !

> -1.0
```

Some examples

- **NumPy**: Basic package for scientific computing. Very fast with mathematical and matrix operations. You can create "ndarrays" which are flexible, efficient and also faster than lists.
- **SciPy**: More advanced than Numpy (e.g. find the determinant or the inverse of a matrix, solve linear equations).
- **Matplotlib**: Plotting data, with complete control over the outline of graphs.
- **Pandas**: Loading datasets and data manipulation.
- **Scikit-learn**: Classification, clustering, basic machine learning

Some examples

- **Requests + BeautifulSoup**: Scraping data from websites
- **NLTK, Regex, Fuzzywuzzy**: Text and natural language processing (NLP)
- **OpenCV**: Images and computer vision (CV)
- **Statsmodels**: Statistical analysis and regressions
- **Tensorflow, Keras, PyTorch**: Advanced machine learning

Loading a dataset

- We will use the **Pandas package** to load datasets
- You can load many software-specific types of files
- Import pandas and find the appropriate command to your dataset:
  - `pd.import_csv()` for comma-separated values (.csv)
  - `pd.import_excel()` for Excel datasets (.xlsx)
  - `pd.import_stata()` for Stata datasets (.dta)
  - `pd.import_r()` for R files (.R)

→ Simply enter the path to your file inside the parentheses

Loading a dataset

- Pandas comes with a special data type to handle datasets: **DataFrames**
- They are very popular for handling and managing tabular data
- Versatile, it can do most of the data cleaning:
  - rename variables, replace or filter values
  - append, merge, collapse rows and columns
- Fast and efficient, up to a few gigabytes (depending on your computer)

Loading a dataset

A short example, using my own research on metaphors:

```
import os # to navigate between paths
import pandas as pd

os.chdir("/home/picard0001/Desktop/python_example")

df = pd.read_csv("data_raw/Alabama_2022.csv")
```

- Here, we use `os.chdir()` to set the working directory
- We capture paths in string format, do not forget " or ' around them

Summary statistics

Before going any further:

- A DataFrame contains **rows** (observations) and **columns** (variables)
- The **dimensions** of the DataFrame can be seen in the data viewer
- Each column has its own data type, use `df.dtypes` in the console to see them all at once
- Columns are usually objects (`object`), which is a special data type

Mea Culpa

While I speak, I tend to use both Python and Stata notations (in parentheses)

Summary statistics

→ Let's have a look at the DataFrame we have opened...

- We access columns using brackets: `df["filename"]`
- We access rows using their index: `df.iloc[1]`
- Subsetting rows in a dataset works just like lists: `df[1:3]`

Summary statistics

Basic summary statistics functions:

Function	Description
`df.dtypes`	Show all data types
`df["metaphor_score"].mean()`	Display the mean of the variable
`df["metaphor_score"].std()`	Display the standard error
`df["metaphor_score"].max()`	Display the maximum value (and so on)
`df["metaphor_score"].describe()`	Display N, mean, std, p10, median...
`df["arg1"].value_counts()`	Tabulate all values and frequencies
`df["speaker"].unique()`	Look for duplicates

Here are nice websites for translating Stata **[Click here]** and R **[Click here]** commands to Python

Data manipulation

We would like to select metaphors in our sample:

```
# Drop the filename column
df = df.drop(columns = ["filename"])

# Rename the state column
df = df.rename(columns = {"st_name": "state"})

# Filter out bad metaphor scores
df = df[df["metaphor_score"] >= 0.7]

# Create a new metaphor column
df["metaphor"] = df["arg0"] + " " + df["arg1"]
```

Apply

If you want to *apply* a rule-based manipulation on all rows, use the `apply` function

```
# Recode the gender variable from int to str
def recode_party(gender):
        gender_str = ""
        if gender == 1:
                gender_str = "Woman"
        else:
                gender_str = "Man"
        return gender_str

df["gender_str"] = df.apply(lambda x: recode_party(x["gender"]),
                                                    axis = 1)
```

Append

A short example for appending datasets:

```
import os
import glob # to store many file names
import pandas as pd

os.chdir("/home/picard0001/Desktop/python_example")

files = glob.glob("data_raw/*.csv") # star = "any"

df = pd.DataFrame() # creates an empty dataframe

for file in files:
        data = pd.read_csv(file)
        df = pd.concat([df, data])
```

`df.append(data)` is deprecated = does not work on latest versions of Pandas !

Merge

We want to merge information on the political party of each speaker

```
df_party = pd.read_csv("political_party.csv")

df_merged = df.merge(df_party, on = "st_name", indicator = True,
                     how = "outer") # or "left", "right", "inner"

# print the output of the merge
print(df_merged['_merge'].value_counts())

> both            13186
> right_only          1
> left_only           0
> Name: _merge, dtype: int64
```

Merge

- Here, we are in a situation where one speaker belongs to one party
- But we have multiple rows for each speaker !
- We can enforce the type of merge using the "validate" parameter:
  - `1:1` = one-to-one
  - `m:1` = many-to-one / `1:m` = one-to-many
  - `m:m` = many-to-many

```
df_merged = df.merge(df_party, on = "st_name", indicator = True,
                     validate = "m:1") # or "many_to_one"
```

Note: Default value of "how" parameter is "inner", so we can omit it here

Collapse

Now, which political party employs the most metaphors ?

We can answer this question by summing metaphors by party

```
df_merged = df_merged[df_merged["metaphor_score"] >= 0.7]
df_merged["nb_metaphors"] = 1

df_collapsed = df_merged.groupby(
                "party", as_index = False)["nb_metaphors"].sum()

print(df_collapsed)

>          party  nb_metaphors
> 0     Democrat           220
> 1   Republican           305
```

Reshape

Now, suppose we want one column by party, then we need to reshape our dataset from long to wide

```
df_collapsed["statistic"] = "metaphor frequency"

df_wide = df_collapsed.pivot(index = "statistic",
                             columns = "party",
                             values = "nb_metaphors")


print(df_wide)


> party                  Democrat  Republican
> statistic
> metaphor frequency          220         305
```

Note: `stack` and `unstack` are elegant substitutes

Plotting data

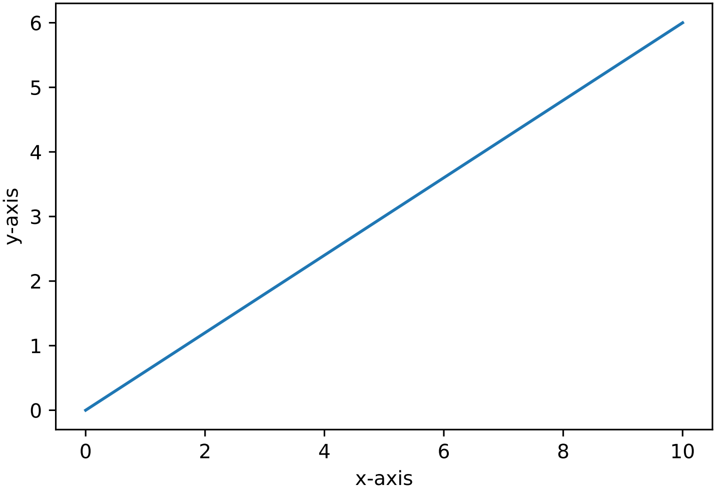
The easiest way to plot data is by using the `plot()` function from Matplotlib

```
import matplotlib.pyplot as plt
import numpy as np

x_vals = np.linspace(0,10,10)
y_vals = np.linspace(0,6,10)

plt.plot(x_vals, y_vals)
plt.ylabel("y-axis")
plt.xlabel("x-axis")
plt.savefig("plot_example.png") # save as png
plt.savefig("plot_example.pdf") # save as pdf
plt.show()
```

Plotting data



Plotting data

Useful Pyplot functions:

Function	Description
`plt.plot()`	Plot y versus x as lines and/or markers
`plt.ylabel()`	Set the label for the y-axis
`plt.xlabel()`	Set the label for the x-axis
`plt.axis()`	Method to get or set some axis properties
`plt.title()`	Set a title for the axes
`plt.scatter()`	A scatter plot of y vs x
`plt.bar()`	Make a bar plot
`plt.figure()`	Create a new figure
`plt.suptitle()`	Add a centered title to the figure
`plt.subplot()`	Add a subplot to the current figure
`plt.show()`	Display the figure

Plotting data

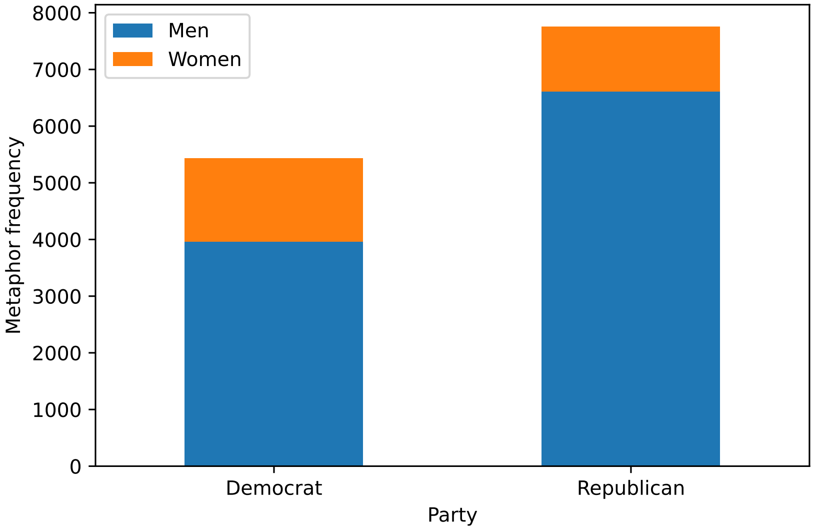
Histograms, pie charts, violin plots... everything is possible !

```
df_bar = df_merged.groupby(["party", "gender"],
                          as_index = False)["nb_metaphors"].sum()

df_bar = df_bar.pivot(index = "party",
                          columns = "gender", values = "nb_metaphors")

ax = df_bar.plot.bar(stacked = True, rot = 0)
ax.set_ylabel("Metaphor frequency"); ax.set_xlabel("Party")
ax.legend(["Men", "Women"])
plt.tight_layout()
plt.savefig("plot_example2.pdf")
plt.show()
```

Plotting data

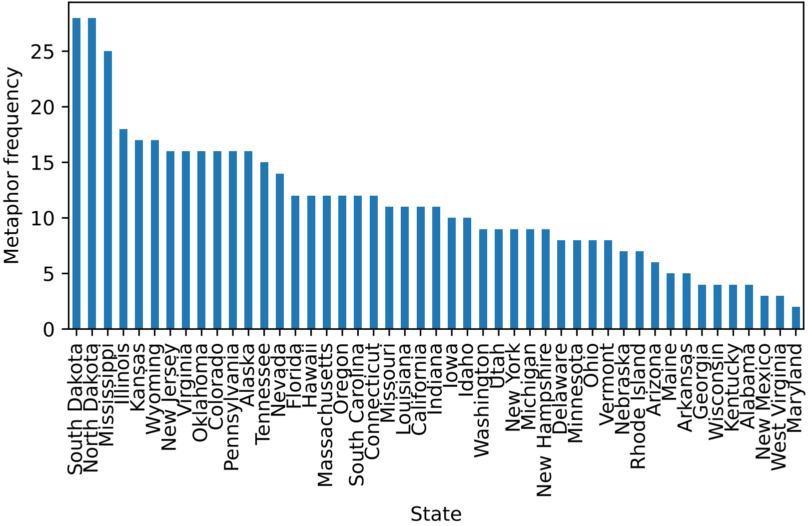


Now it's your turn !

Find out which U.S. state uses the most metaphors:

1. Append all datasets from the folder `data_raw`
2. Clean the columns of interest
3. Collapse the dataset to get the number of metaphors by state
4. Plot metaphor frequencies by state in a nice histogram

Example


Metaphor frequency
0
5
10
15
20
25
State
South Dakota
North Dakota
Mississippi
Illinois
Kansas
Wyoming
New Jersey
Virginia
Oklahoma
Colorado
Pennsylvania
Alaska
Tennessee
Nevada
Florida
Hawaii
Massachusetts
Oregon
South Carolina
Connecticut
Missouri
Louisiana
California
Indiana
Iowa
Idaho
Washington
Utah
New York
Michigan
New Hampshire
Delaware
Minnesota
Ohio
Vermont
Nebraska
Rhode Island
Arizona
Maine
Arkansas
Georgia
Wisconsin
Kentucky
Alabama
New Mexico
West Virginia
Maryland

Asking for help

Where you can find help

The documentation: Every package comes with a document for each function, containing information on:

- What the function does
- A list of arguments, and what they are
- Some examples on how to use them

Specialized websites: A great source of questions and answers (Stack Overflow mainly...)

Where you can find help

Search engines: Google, Yahoo, Yandex... Another way to find answers (tutorials, videos, short courses)

- Lot of content, but very few is applicable to your own *special* question
- Answers usually outdated, or simply not be the best anymore
- ChatGPT ? Yes but careful of copy-pasting, mistakes happen !

Friends and university staff: sharing your questions with someone also helps:

- Short questions can be answered very fast
- They may learn from your questions as well
- ...but their time is limited !

Where you can find help

Whenever possible, you should try to follow this rule of thumb:

- First, read the documentation
- Second, browse websites such as Stack Overflow
- Only then, use a search engine
- Lastly, ask friends, then university staff

What are you looking for ?

- *"I don't know how to code something"*
  - Structure your question with a few keywords
  - Look for answers online
  - If none apply to your question, ask your friends or on e.g. Stack Overflow
- *"I tried something but my code doesn't give me the expected result"*
  - Be careful of copy and pasting things online, review your code
  - If you are using a function/package, refer to the documentation of that package
  - If not, troubleshoot your code: follow what it does line by line and verify that is gives you what you want using a simple model (e.g. fake data)

What are you looking for ?

- *"I don't know how to code something"*
  - Structure your question with a few keywords
  - Look for answers online
  - If none apply to your question, ask your friends or on e.g. Stack Overflow

- *"I tried something but my code doesn't give me the expected result"*
  - Be careful of copy and pasting things online, review your code
  - If you are using a function/package, refer to the documentation of that package
  - If not, troubleshoot your code: follow what it does line by line and verify that is gives you what you want using a simple model (e.g. fake data)

What are you looking for ?

- *"I don't know how to code something"*
  - Structure your question with a few keywords
  - Look for answers online
  - If none apply to your question, ask your friends or on e.g. Stack Overflow
- *"I tried something but my code doesn't give me the expected result"*
  - Be careful of copy and pasting things online, review your code
  - If you are using a function/package, refer to the documentation of that package
  - If not, troubleshoot your code: follow what it does line by line and verify that is gives you what you want using a simple model (e.g. fake data)

What are you looking for ?

- *"I don't know how to code something"*
  - Structure your question with a few keywords
  - Look for answers online
  - If none apply to your question, ask your friends or on e.g. Stack Overflow
- *"I tried something but my code doesn't give me the expected result"*
  - Be careful of copy and pasting things online, review your code
  - If you are using a function/package, refer to the documentation of that package
  - If not, troubleshoot your code: follow what it does line by line and verify that is gives you what you want using a simple model (e.g. fake data)

What are you looking for ?

- *"My code doesn't run"*
  - The console is your ally, search for the line number at which the code breaks
  - Read the error message and try to understand what it means
  - If the message isn't clear, copy and paste it on a search engine
  - Pay attention to the data types, sometimes they are incompatible
  - If you are using a function/package, refer to the documentation of that package
  - If the problem lies inside a loop, try to solve it outside of the loop

→ General rule: try to break down the problem; identify the source and make it run alone, then add it back to your code.

What are you looking for ?

- *"My code doesn't run"*
  - The console is your ally, search for the line number at which the code breaks
  - Read the error message and try to understand what it means
  - If the message isn't clear, copy and paste it on a search engine
  - Pay attention to the data types, sometimes they are incompatible
  - If you are using a function/package, refer to the documentation of that package
  - If the problem lies inside a loop, try to solve it outside of the loop

→ General rule: try to break down the problem; identify the source and make it run alone, then add it back to your code.

What are you looking for ?

- *"My code doesn't run"*
  - The console is your ally, search for the line number at which the code breaks
  - Read the error message and try to understand what it means
  - If the message isn't clear, copy and paste it on a search engine
  - Pay attention to the data types, sometimes they are incompatible
  - If you are using a function/package, refer to the documentation of that package
  - If the problem lies inside a loop, try to solve it outside of the loop

→ General rule: try to break down the problem; identify the source and make it run alone, then add it back to your code.

Stack Overflow: How it works

- This website prioritizes quality over quantity of questions (or "posts")
- Do not ask a question before checking if it has already been answered before
- Only after, ask your question in the clearest and shortest way
  - Focus on what you don't know, skip all the details that you know how to do
  - Explain what you have tried before
  - Add a reproducible example (some code with fake data)
  - End your post by writing what the outcome should look like

→ Link to all the rules: `https://stackoverflow.com/help/how-to-ask`

Stack Overflow: Some examples

Badly written questions:

- `https://stackoverflow.com/questions/78106125/how-bypass-kleinanzeigen-js-detected-input-in-email`
- `https://stackoverflow.com/questions/75506603/pandas-combine-two-if-one-is-empty`

Nicely written questions:

- `https://stackoverflow.com/questions/75502195/validate-string-format-based-on-format`
- `https://stackoverflow.com/questions/75505923/how-to-skip-2-data-index-array-on-numpy`

Stack Overflow: Careful !

- Usually not the fastest way to answer your question: you could get a response in minutes, but most of the time it takes a few days (if anyone dares to help!)
- People won’t try to be nice to you (no need to say "hi" and "thanks" as well)
- People might misunderstand your question, or tell you why you shouldn’t do it this way
- People might give you a solution that works for the example you’ve laid out to them, but not on your real dataset (different data, issues of scale...)

Wrapping-up

Wrapping-up

With this course, you should now be able to:

- Install Python, set-up your first environment
- Understand most data types and work with them
- Load packages and datasets, perform basic data manipulation
- Efficiently look for help in the future...

Questions, remarks ?

leo.picard@unibas.ch