Python meeting

Python meeting 9

10th June 2026

The Python Meeting will start soon

COURSE PYTHON SCHEDULE

01: 20250114, 9am: Initiation – Case-study: P02_15: Above average prices

- Set-up: Python, Visual Studio Code
- Project creation: Folder, .py, Virtual Environment, CMD
- Libraries: Import, requirements file
- Naming conventions
- Basic objects: Dataframe (table), Series (column), Variable, Function, Group_by
- Basic functions: Import, Rename, Filter, Join, Create columns, Group, Aggregate, Export

02: 20250121,10am: Melt, split, for – Case-studies: D01: Trial balance, P01_19: Supplier debtors, O01_19: Customer creditors

- Using melt, split
- For loop

03: 20250128, 9am: Web-download – Case-study: P01_10: Suppliers in OFAC

- Request for downloading sanctions
- IO for encoding
- Regex for comparison
- Split, explode for row generation
- Levenshtein distance scoring

04: 20250204, 9am: JSON, expressions, rolling Window – Case-study: P02_19: Split POs

- Import JSON, Expression object for adding labels,
- cum_sum() for rolling window

05: 20250211, 9AM: zip, dynamic fields – Case-study: P02_03: PO whilst blocked

- Zip to group lists together
- Using dynamic fields in a for loop

COURSE PYTHON SCHEDULE

06: 20250218, 10am: Cumulative sum, dates – Case-study: P02_18: PO slow rotation

- cum_sum for calculating future or past inventory movements

07: 20250225, 9am: Isolation forest for unusual transactions – Case-study: Unusual bank transfers

- Using isolation forest library to score for unusual transactions

08: 20250304: 9am: SpaCy – Case-study: P01_11/ O01_11: Suppliers/ customers that are people

- Using SpaCy NLP object to categorize names

09: 20250311: 9am: Contract review – Case-study: Scoring of contracts

- Using Gemini model to check for paragraphs of similar meaning

10: 20250318: 10am: Open AI API – Case-study: Generating audit reports

- Using API to OpenAI to generate text for audit reports

11: 20250325: 9am: Regex and Sklearn matching – Case-study: HR03_12/ H403_13: Unusual T&E

- Using Regex and Sklearn to check for unexpected travel and expenses

12: 20250401: 9am: OCR: Image recognition – Case-study: HR03_14: Expenses for others

- Using OCR python library to read travel and expense receipts

TODAY'S SCHEDULE

01 *What is an LLM?*

02 *Connecting to an LLM with python*

03 *Running contract analysis using LLMs*


01
What is an LLM?

Poll Question

What do you use LLMs for?

20260610_01_WhatDoYouUseLLMsFor?

1. What do you use LLMs for? (Single choice)

- ○ Reading and summarizing audit reports
- ○ Reading and scoring contracts
- ○ Summarizing data about 3rd parties
- ○ Other? (Mention in chat)

Automate research and documentation with LLMs
How do we decide which model to use?

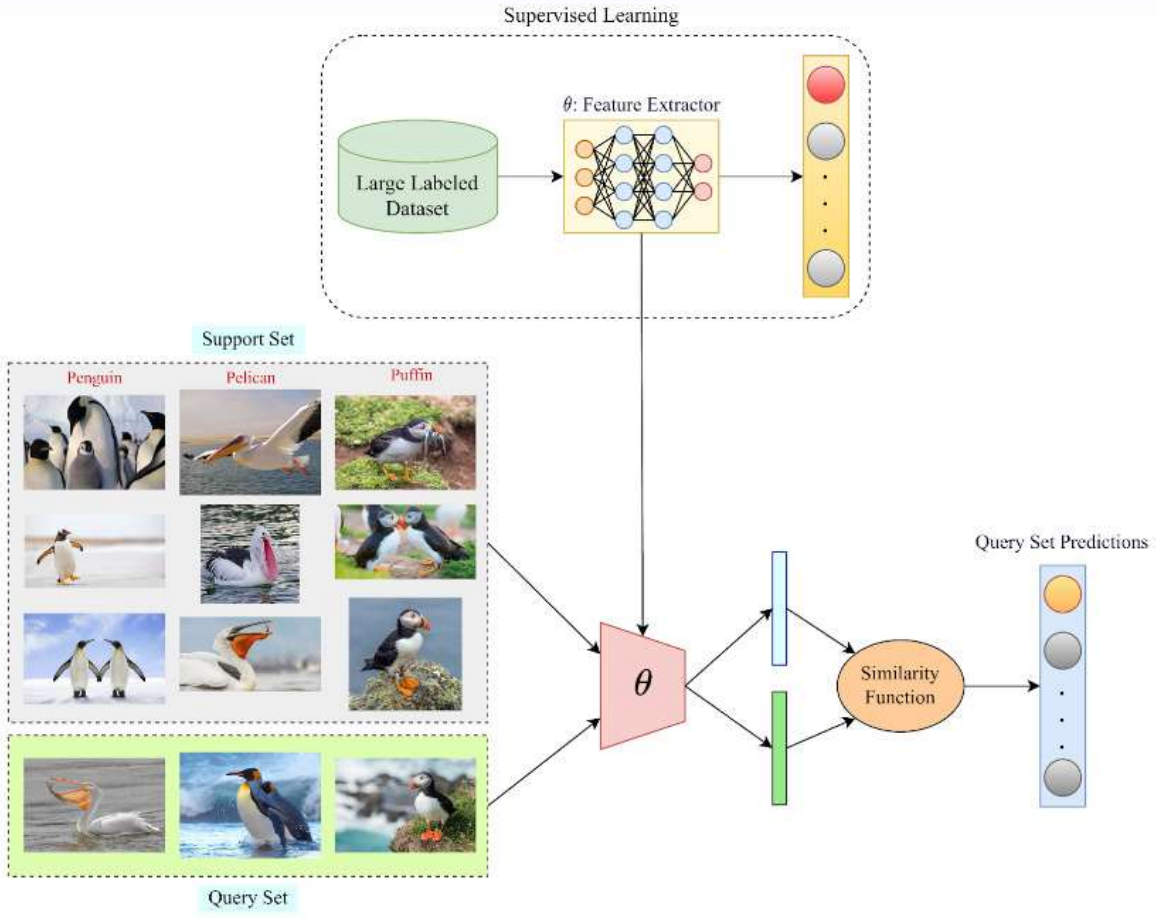
- There are many LLM models out there – they get grades – like at school!
- https://paperswithcode.com/sota/multi-task-language-understanding-on-mmlu?tag_filter=0%2C183%2C184%2C238%2C318%2C188%2C567

MMLU: Massive Multi-task Language Understanding:

- Benchmark giving scores based on:
  - **Few shot**
  - **Zero shot**
- Scores are created based on problems from 57 subject areas
- Levels are from elementary-level to professional-level problem

Automate research and documentation with LLMs
What is few shot?

- What is Few Shot?
- https://blog.paperspace.com/few-shot-learning/#:~:text=Few%2DShot%20Learning%20(FSL),learning%20means%20learning%20to%20learn).



Few Shot:

Ability to see a FEW examples and then classify correctly a new example that was not seen before.

Example, go to the zoo, see some birds.. Then see a picture of a bird and be able to guess if it is a pelican or a penguin.

Automate research and documentation with LLMs
What is zero shot?

- What is Zero Shot?

Obviously similar!

If a child had never seen a zebra or picture of a zebra before – they would say "Oh look – a Stripey Horse!"

Why?

- 4 legs
- Arched back
- Tail
- Mane
- Pointy ears

Zero Shot:

Ability to see something completely new and recognize that it is similar to something in training.

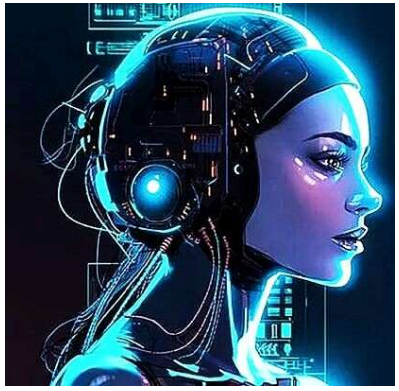
Example, zebras not included in pre-training – but we still can recognize that it looks like a stripey horse.

Machine Learning works in similar ways

It can compare all the attributes of everything it has learned in pre-training and compare to something new in order to make an association.

Automate research and documentation with LLMs
What is Gemma?

- What is Gemma? (https://blog.google/technology/developers/gemma-open-models/#:~:text=Gemma%20is%20a%20family%20of,%2C%20meaning%20%E2%80%9Cprecious%20stone.%E2%80%9D)
  - Gemma is a “Light-Weight” version of Gemini
  - It is a model that is pre-trained on trillions of pieces of information
  - It comes in 2B and 7B versions
  - Because it is lighter than Gemini, it is not as powerful as Gemini, but for simple everyday tasks it is good enough… and it can run on a normal machine



Greater pre-training on wider topics/ Do extremely hard reasoning/ maths/ etc.

Rival to ChatGPT.

Gemini: requires massive RAM power

Gemma

Do everyday reasoning on everyday topics!

It’s FREE

Gemma: you can use it on your local PC!

Automate research and documentation with LLMs
Checking the performance of light-weight models

How does Gemma stack-up as a lightweight model?

- Quite good MMLU scores
- Quite good reasoning – especially for commonsense reasoning/ everyday tasks
- Basic maths… kind of ok
- Generation of code from human sentences – not amazing

CAPABILITY	BENCHMARK	DESCRIPTION	Gemma 7B	Llama-2 7B	Llama-2 13B
General	MMLU 5-shot, top-1	Representation of questions in 57 subjects (incl. STEM, humanities and others)	64.3	45.3	54.8
Reasoning	BBH -	Diverse set of challenging tasks requiring multi-step reasoning	55.1	32.6	39.4
	HellaSwag 0-shot	Commonsense reasoning for everyday tasks	81.2	77.2	80.7
Math	GSM8K maj@1	Basic arithmetic manipulations (incl. Grade School math problems)	46.4	14.6	28.7
	MATH 4-shot	Challenging math problems (incl. algebra, geometry, pre-calculus, and others)	24.3	2.5	3.9
Code	HumanEval pass@1	Python code generation	32.3	12.8	18.3

https://blog.google/technology/developers/gemma-open-models/#:~:text=Gemma%20is%20a%20family%20of,%2C%20meaning%20%E2%80%9Cprecious%20stone.%E2%80%9D

Poll Question

What do you think makes a good model?

20260610_02_WhatDoYouThinkMakesAGoodModel?

1. **What do you think makes a good model?** (Multiple choice)

- [ ] Can reply fast
- [ ] Is accurate and can give a response in a specific format
- [ ] Is not too expensive - no point spending too much for simple tasks
- [ ] Is set up with an API that has documented instructions that can be adjusted in order to guide the user
- [ ] Is easy to use - simple interface - easy to login to Single Sign On possible


02
Connecting to an LLM with python

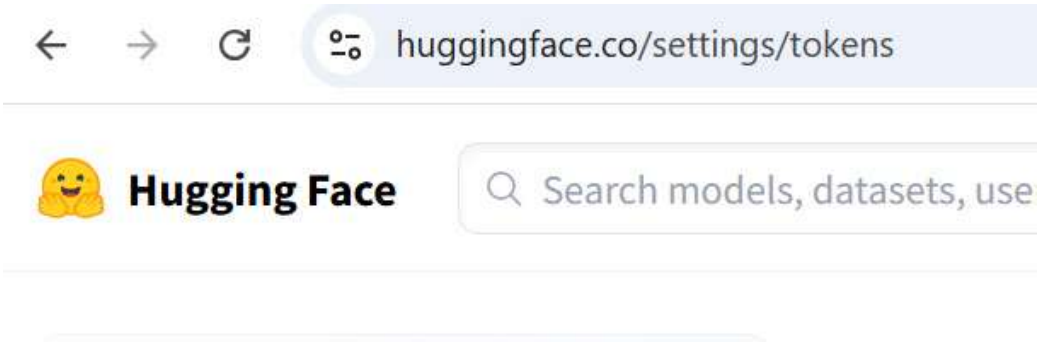
Hugging Face: GitHub for free AI Models

- In order to use a Gemini model, we can create an account on Hugging Face.
- Hugging Face is a site that enables you to use Gemini models for free!
- These models are pre-trained.

Hugging Face is an AI company and an open-source ecosystem that provides tools, models, and datasets for **natural language processing (NLP)**, **computer vision**, **speech**, and now even **multimodal** and **code** tasks.

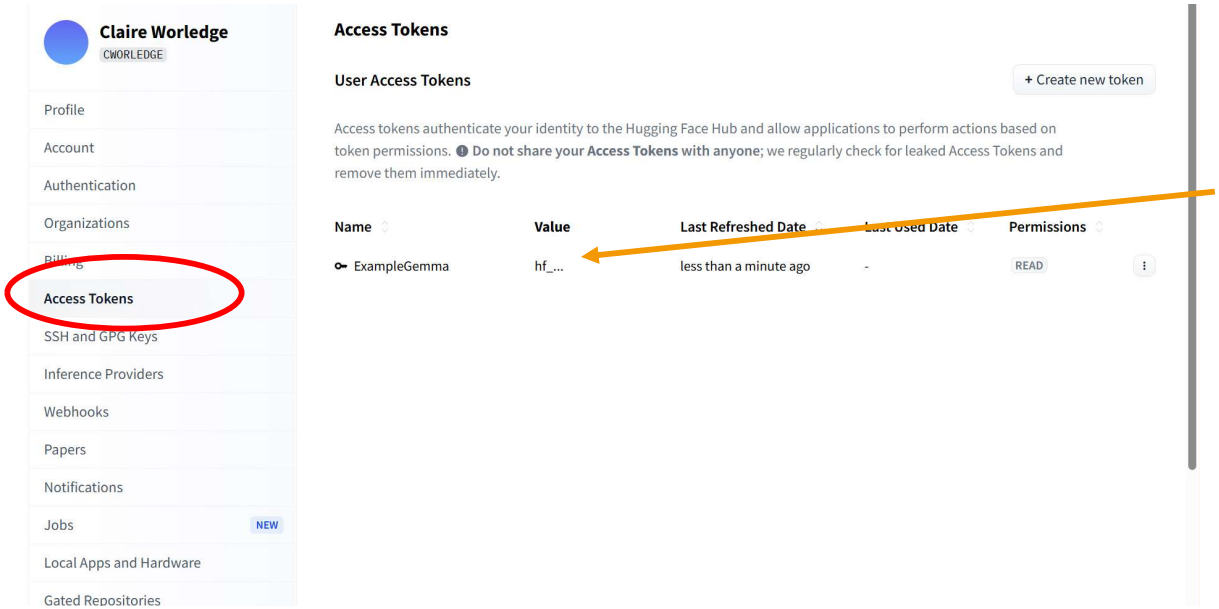
🧠 **In simple terms**

It's like **GitHub for machine-learning models** — a huge public hub where developers share, download, and run pretrained AI models.



https://huggingface.co/settings/tokens

Create a HuggingFace account and API key

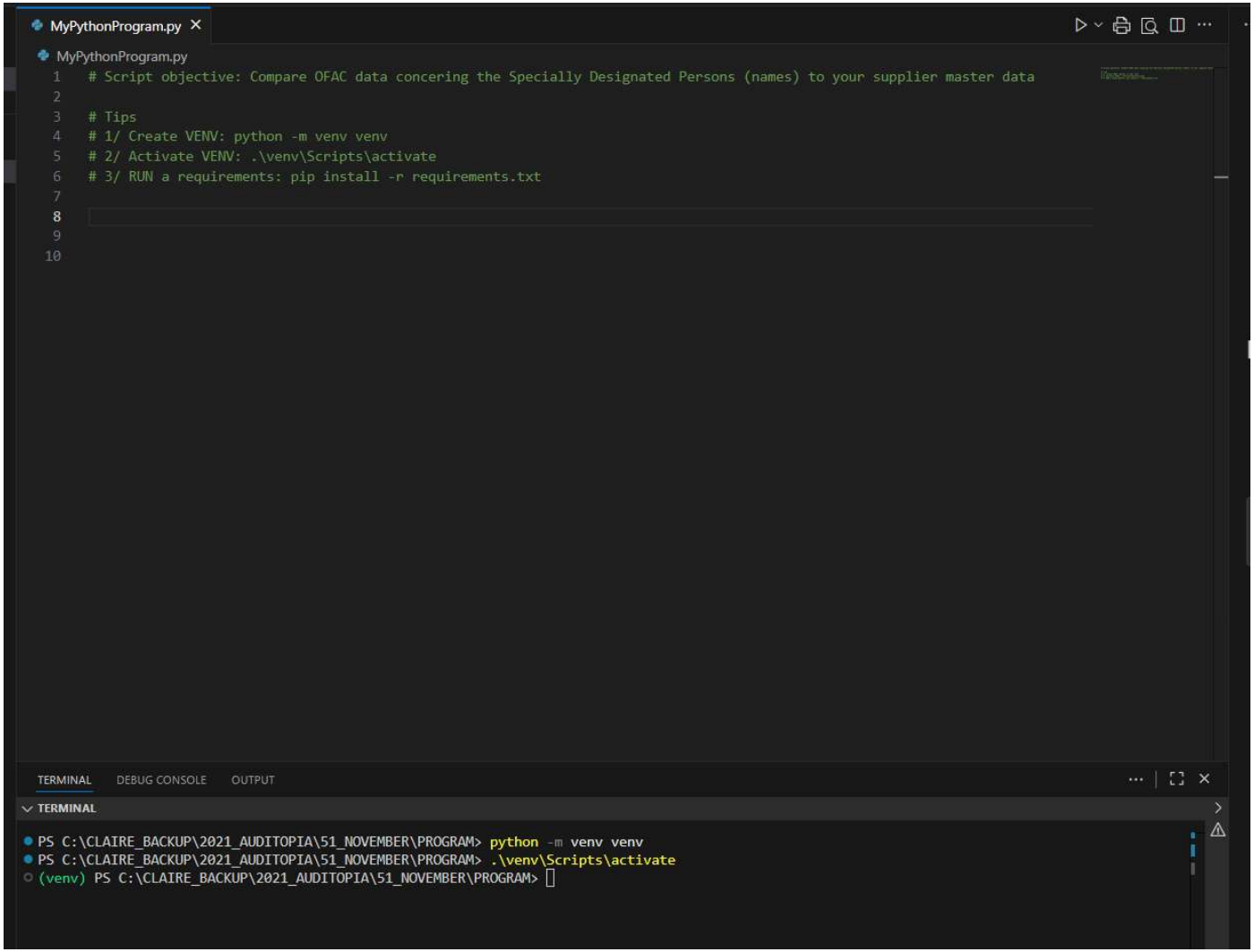


- Once we have created an account, we need to create a token.
- We will use the token in our python script.
- The token enables us to download models from hugging face.

Example simple Python code for connecting with Hugging Face

- We could create a python script, to download a model.
- First open an empty .py script in Visual Studio Code.

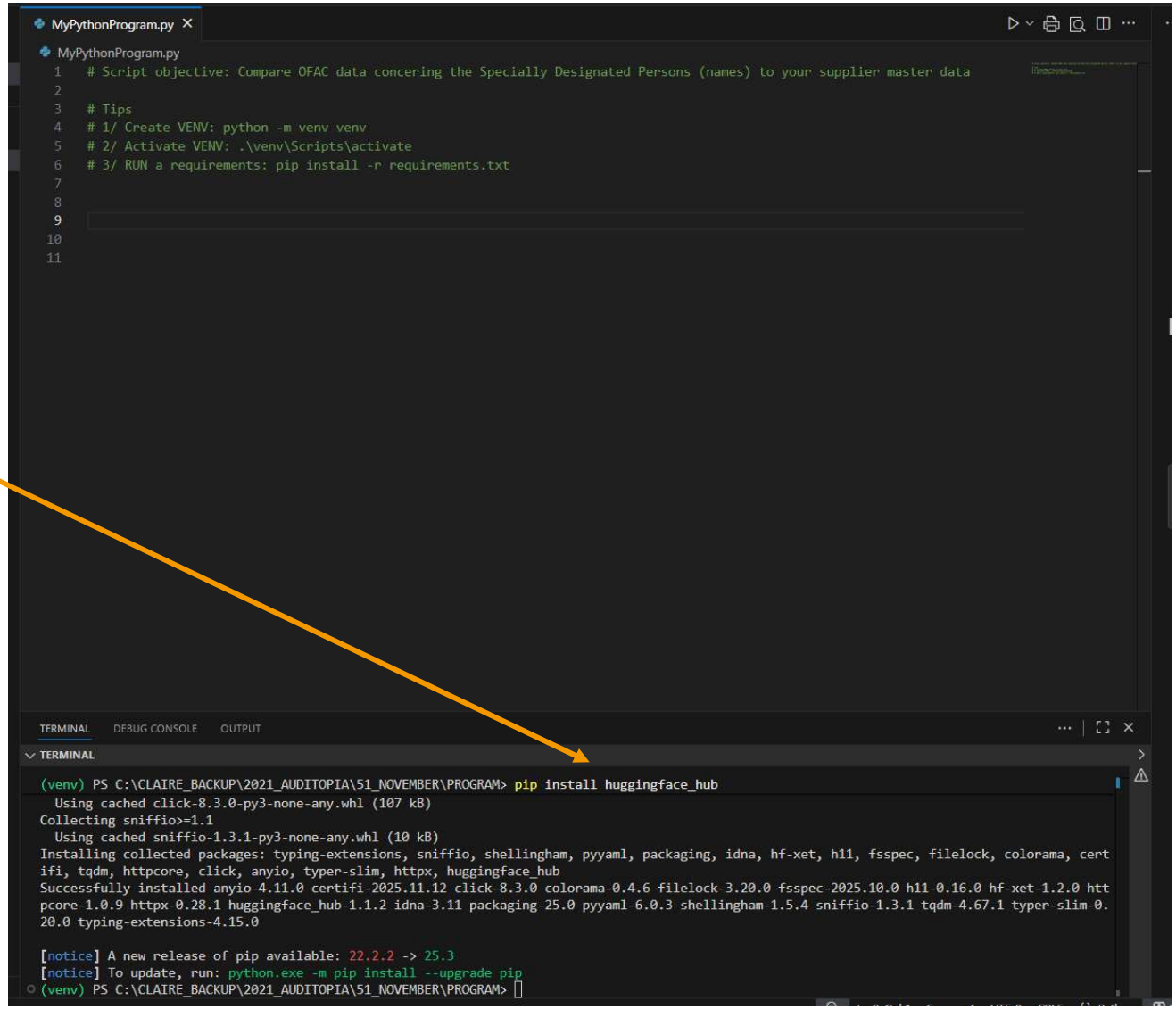
- Then create a virtual environment
- Then activate your virtual environment



Example simple Python code for connecting with Hugging Face

- Then install the python library that lets you talk to the Hugging Face website

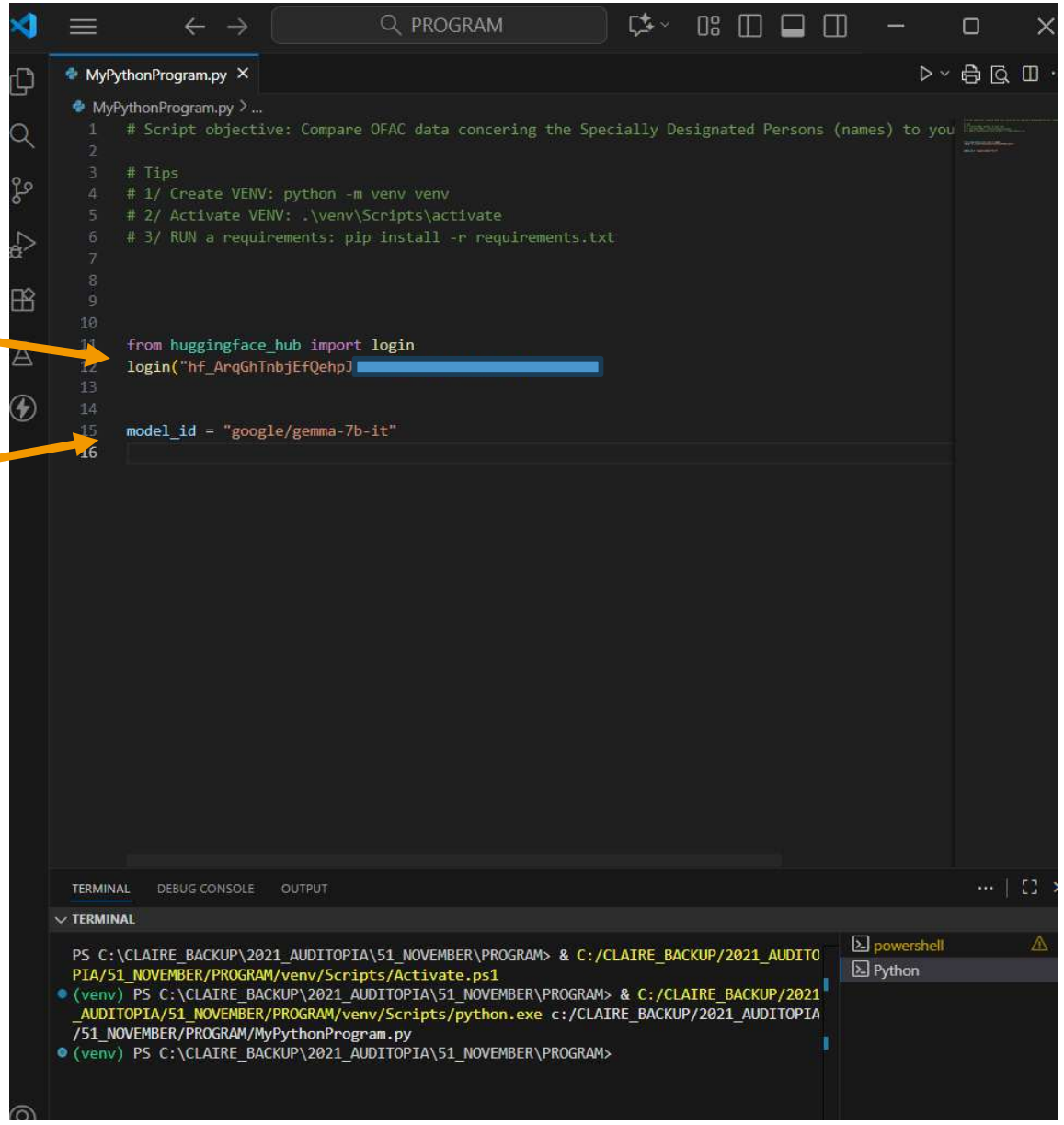
- pip install huggingface_hub



Example simple Python code for downloading an AI model

- With the huggingface_hub library installed in our virtual environment, next we can login to the hugging face website from python, using our token.

- Then we can reference and use models created and trained by amazing Artificial Intelligence experts!



Differences between the PC and Kaggle

- In Python Meeting 09, we have made a version of the project that can run either on a Windows PC, or on Kaggle.

- Your PC:
  - Windows,
  - Visual Studio Code
  - .py files
  - You can choose the Python version when you create the virtual environment.
  - Path addresses are C:\...\YourProject
  - We can easily update the AM_VARIABLES.txt from within Windows or Visual Studio Code

- Kaggle:
  - Linux
  - Jupyter
  - .ipynb files.
  - You cannot choose the Python version when you turn on the Notebook (see later)- at time of writing it is 3.12.
  - Path addresses are \Kaggle\Input for datafiles and \Kaggle\Working for results files
  - If we want to update the source files in Kaggle, we have to upload a new Dataset and then replace our Inputs with it

Making the Python script compatible with Windows PC and Kaggle environments

- In order to be able to test our Python project on a Windows PC before running it for real on a Kaggle machine, we need to make our Python code compatible with both environments.
- To know if we are in KAGGLE:
    - Add ZV_BO_IS_KAGGLE: to indicate if we are in Kaggle
- To find the folder paths:
    - Kaggle: use PI_OS.walk() to find the 02_PROGRAMMES folder
    - Windows: use PI_OS.path.dirname(PI_OS.path.abspath(__file__)) to find the 02_PROGRAMMES folder
    - Use PI_OS.path.dirname(ZV_ST_02PROG_FOLDER) to find the root folder
    - Use PI_SYS.path.append(ZV_ST_02PROG_FOLDER) to add the programmes folder to the system path (this enables us to then import from Z_SHARED_FUNCTIONS).
    - Use PI_OS.path.join(ZV_ST_ROOT_FOLDER,'01_SOURCES') to automatically create sources folder.
- To install libraries: (FC_INSTALL_REQUIREMENTS.txt)
    - The manual pip install is different on Kaggle, so we prefer to use subprocess to run the installations, for simplicity
- Because Kaggle does not support .env files same way as Windows: (FC_CREATE_DI_VARIABLES)
    - Use rglob() to automatically find .env and AM_VARIABLES file paths
    - Add the variables and the AM_VARIABLES into one ZV_DI_VARIABLES dictionary
- Because Kaggle exports to its own Working Directory and not to a Windows folder:
    - Create the 03_RESULTS directory depending on whether or not Kaggle is being used

Enabling testing: small model and truncated text

- To ensure that our project can work first time, when we start using the GEMMA model, we may want to test it first. This is because, running GEMMA takes about ten minutes. We have therefore added two test modes in AM_VARIABLES:

P00_01_CONTRACT_REVIEW.py 9+, M | AM_VARIABLES.txt

CONTRACT_REVIEW > 01_SOURCES > AM_VARIABLES.txt

```
1  ZV_ST_MODEL_ID=google/gemma-3-1b-it
2  ZV_ST_CATEGORIES='Contract Date,Effective Date,Renewal Term
3  ZV_ST_TEST_MODE=False
4  ZV_ST_TEST_MODE_WO_LLM=False
5  ZV_ST_RESULTS_FILE=ContractReviewResults.xlsx
```

We can set ZV_ST_TEST-MODE = True to use a small model and just part of one file.

```
# 3/ Test mode - overwrite variables
if ZV_BO_TEST_MODE:
    ZV_ST_MODEL_ID = 'sshleifer/tiny-gpt2'
    ZV_LI_CATEGORIES = ['Payment terms']
    ZV_NU_MAX_FILES = 1
    ZV_NU_MAX_CHARS = 2000
```

```
if ZV_BO_TEST_MODE:
    ZV_LI_LI_INPUT_TOKEN_IDS = (
        ZV_OB_TOKENIZER
        .encode(
            ZV_ST_CHAT_TEMPLATE,
            add_special_tokens=False,
            return_tensors='pt',
            truncation=True,
            max_length=512
        )
        .to(ZV_OB_LLM_MODEL.device)
    )
```

```
# Test mode
if ZV_BO_TEST_MODE:
    ZV_ST_CONTRACT_TEXT = ZV_ST_CONTRACT_TEXT[:ZV_NU_MAX_CHARS]
```

With ZV_ST_TEST_MODE = True, the code:

- changes the model to a small one
- reduces category list to one category
- truncates the list of tokens
- only looks at one file
- truncates the text of the file

Enabling testing: no model

- The second test mode option enables us to run the program without using a model at all. This is useful if we want to be able to just test the import/ extract text from .pdf, .docx, .doc and export,... before we test the use of a model, which can take time to run.

P00_01_CONTRACT_REVIEW.py 9+, M | AM_VARIABLES.txt

CONTRACT_REVIEW > 01_SOURCES > AM_VARIABLES.txt

```
1  ZV_ST_MODEL_ID=google/gemma-3-1b-it
2  ZV_ST_CATEGORIES='Contract Date,Effective Date,Renewal Term,
3  ZV_ST_TEST_MODE=False
4  ZV_ST_TEST_MODE_WO_LLM=False
5  ZV_ST_RESULTS_FILE=ContractReviewResults.xlsx
```

We can set ZV_ST_TEST_MODE_LLM = True to not use a model (just tests library installations, import/ export)

```
if ZV_BO_TEST_MODE_WO_LLM:
    ZV_OB_TOKENIZER = None
    ZV_OB_LLM_MODEL = None
```

```
if ZV_BO_TEST_MODE_WO_LLM:
    ZV_DI_RESPONSE = {
        ZV_ST_CATEGORY: 'TEST RESPONSE'
        for ZV_ST_CATEGORY in ZV_LI_CATEGORIES
    }
else:
    ZV_DI_RESPONSE = FC_GENERATE_RESPONSE(
        ZV_ST_CONTRACT_TEXT,
        ZV_LI_CATEGORIES
    )
```

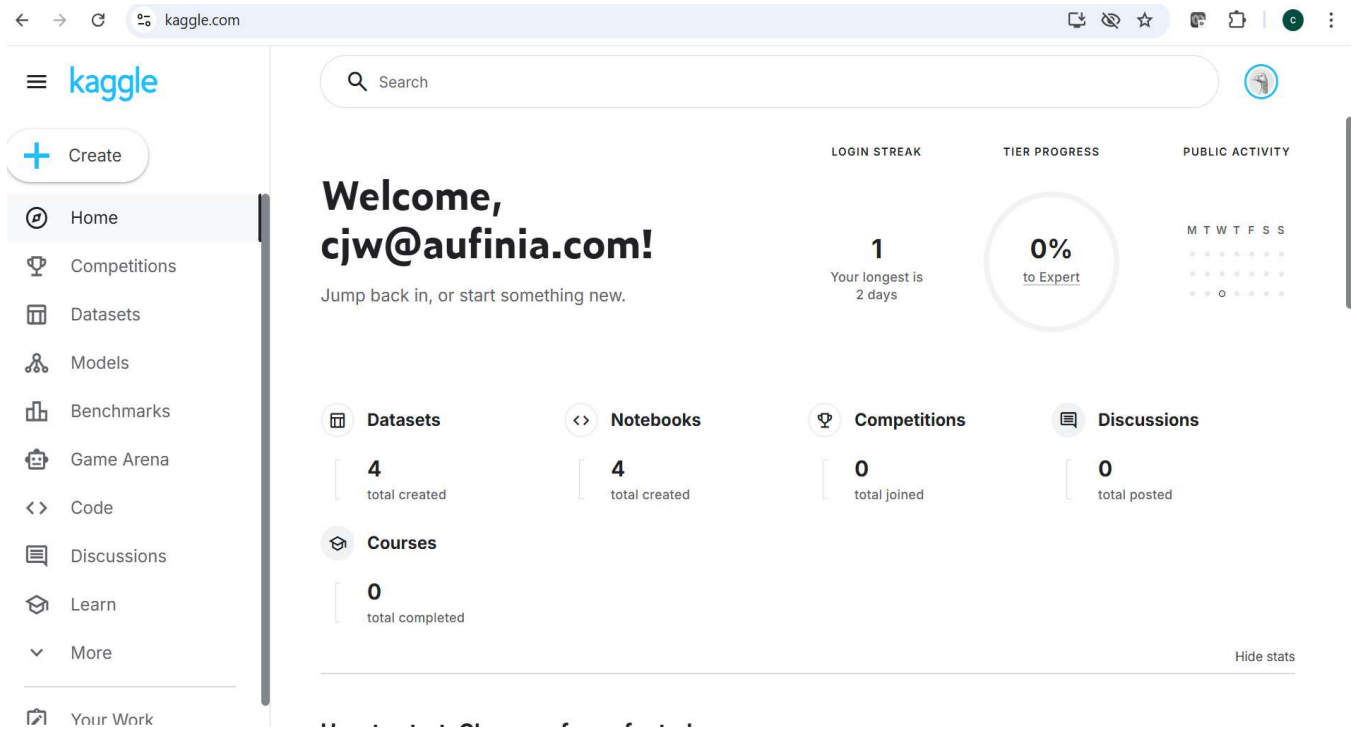
With ZV_ST_TEST_MODE_WO_LLM = true, the code:

- does not get the tokenizer
- does not download the LLM model
- uses a dummy response instead of calling the FC_GENERATE_RESPONSE() function

Automate research and documentation with LLMs
Set-up a Kaggle account

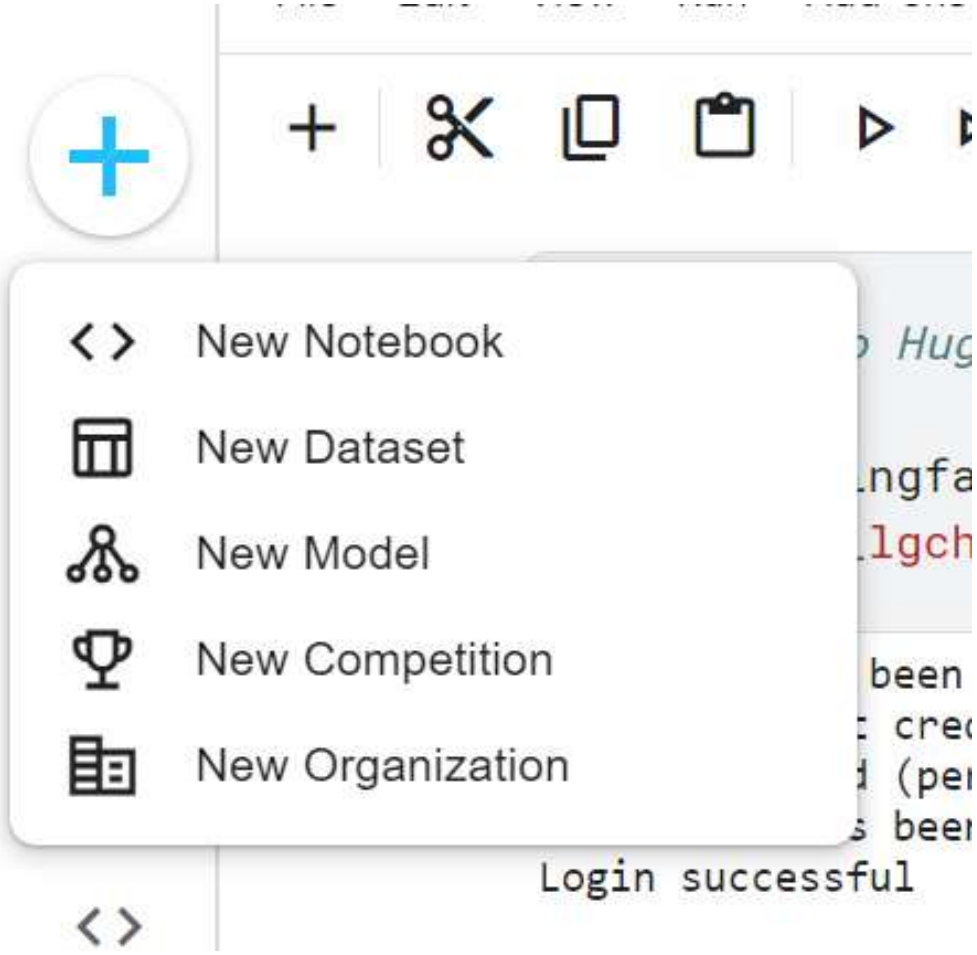
- Running NLP models on a laptop can be slow if the model is large and you do not have a lot of disk-space/ RAM/ GPU etc!
- Optionally, we can run it in Kaggle (if our data is not too sensitive).
- You can create a free Kaggle account here:

www.Kaggle.com



Kaggle: create a new Notebook

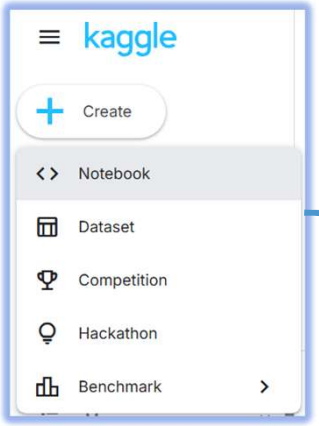
- Create a new Notebook

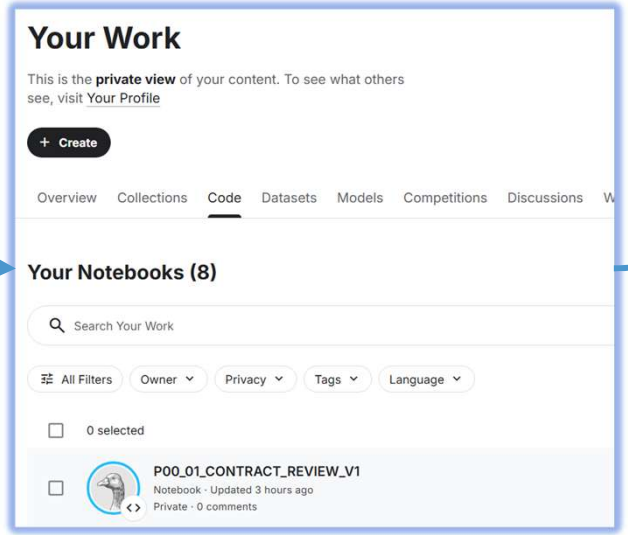


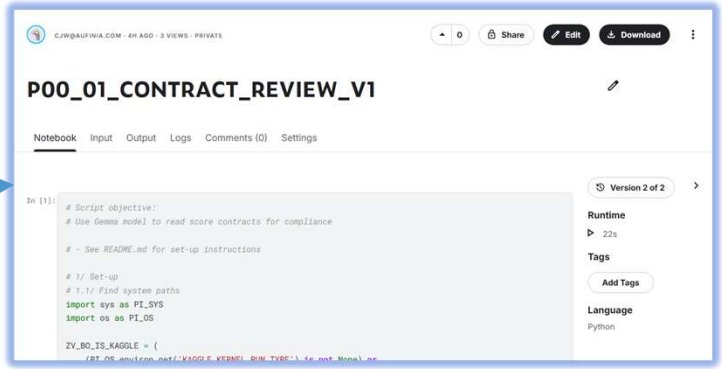
Kaggle: Create a new Notebook and copy .py code to it

If you are used to working in Visual Studio Code, instead of Jupyter, you can add a new notebook and then paste into the notebook the code from your .py file. In this case, we choose Create Notebook, rather than File-> Import Notebook

- Click on + Create and choose Notebook.
- Give the Notebook the name of the main python file: P00_01_CONTRACT_REVIEW
- Then click on the Notebook. If you cannot find it, go to Your Work -> Code and click on it.
- With the Notebook open, paste the code from your Visual Studio Code P00_01_CONTRACT_REVIEW.py into the Notebook in Kaggle.

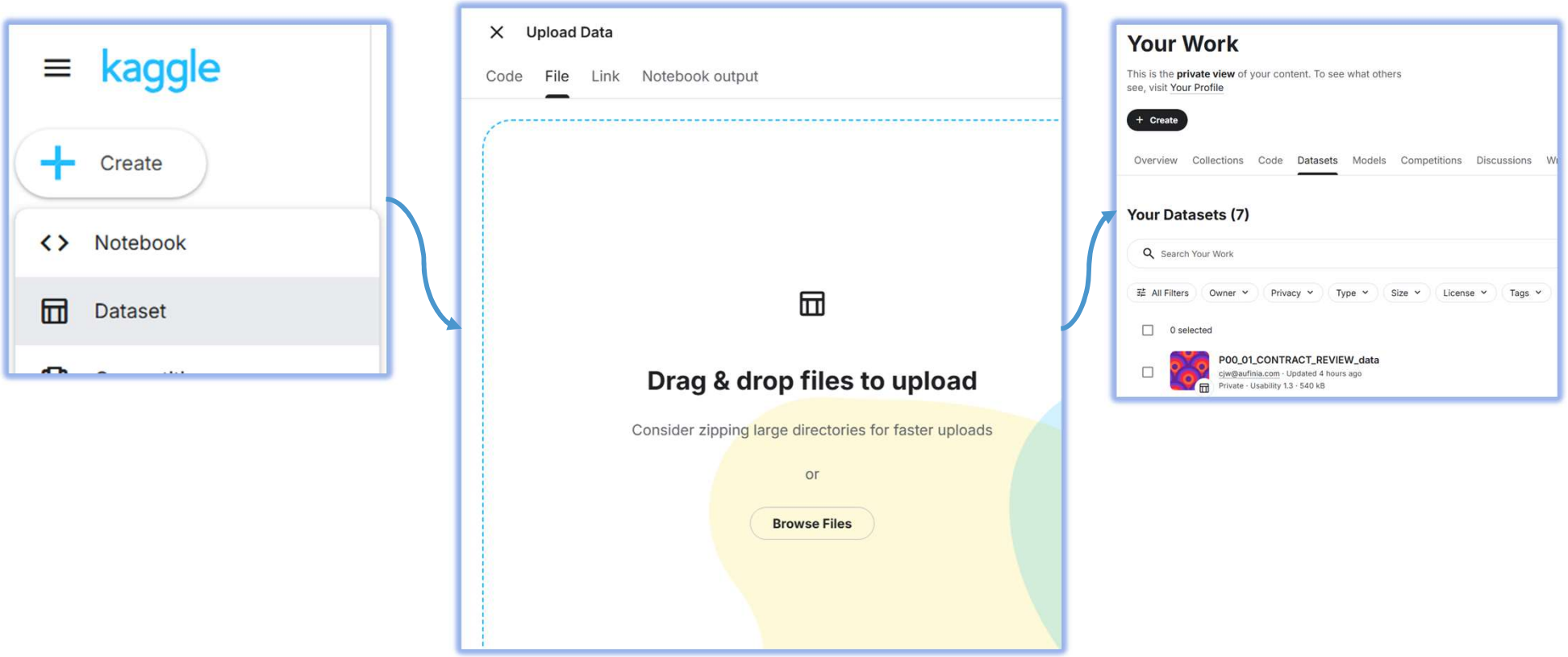






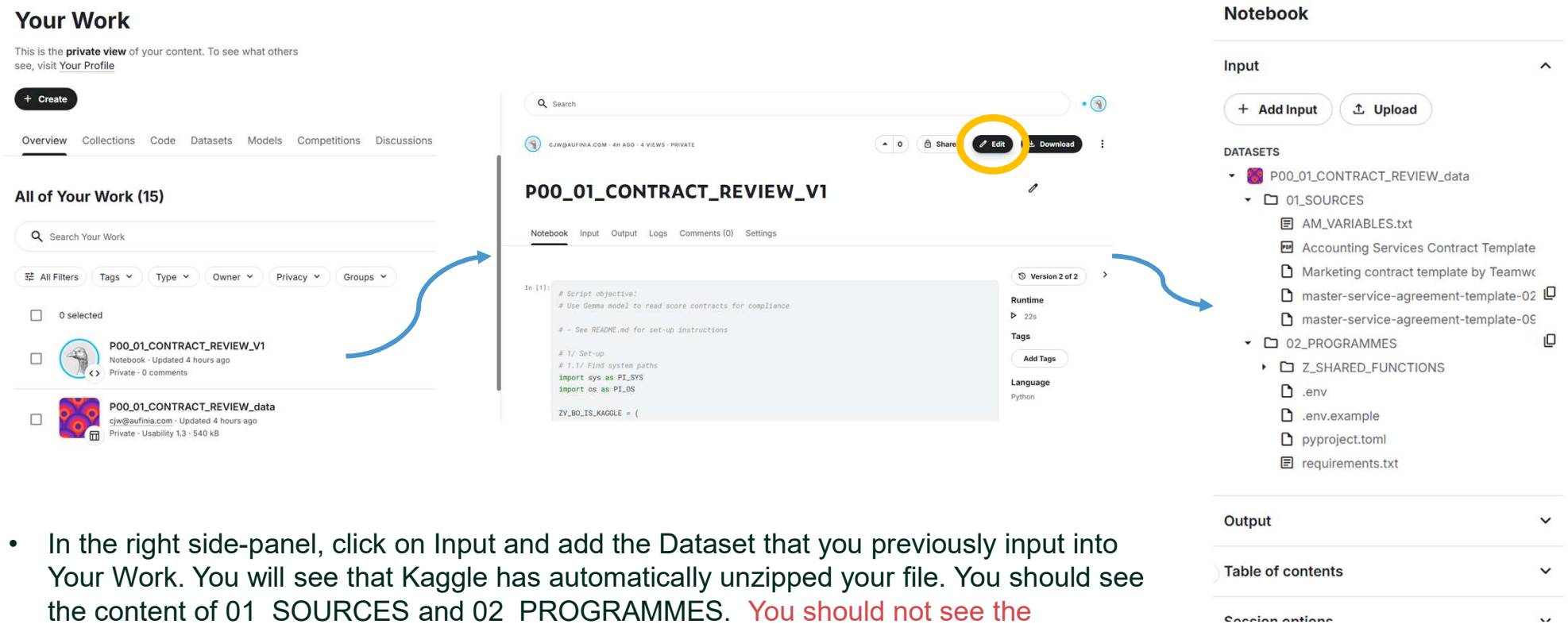
Kaggle: Import a dataset

- In Windows, **remove the P00_01_CONTRACT_REVIEW.py, __pycache__ and /venv from 02_PROGRAMMES folder**., then zip the 02_PROGRAMMES folder and 01_SOURCES folder into one .zip file together.
- Click on + Create and choose Dataset.
- Upload your .zip file and give it the name P00_01_CONTRACT_REVIEW_data
- Kaggle will automatically unzip the dataset and save it in Your Work-> Datasets



Kaggle: Add the data set to the Notebook

- So far, we have created a Notebook and a Dataset. These two things can be seen in Your Work-> Overview. You can also see them under the tabs Your Work: Code and Datasets.
- Click on the Notebook from Your Work that you just created.
- Click on Edit in the Notebook screen to show the right side-panel. Here, we will link to the Dataset and configure the machine for running the script.



- In the right side-panel, click on Input and add the Dataset that you previously input into Your Work. You will see that Kaggle has automatically unzipped your file. You should see the content of 01_SOURCES and 02_PROGRAMMES. You should not see the P00_01_CONTRACT_REVIEW.py, that we previously removed from 02_PROGRAMMES before creating our zip file.

Kaggle: Update the Session options

If you do not see the right-side panel, in Kaggle, go to Home, and then go to Your Work, click on the Notebook you created. You should then see an Edit button. Click on Edit. Then you will see the right-panel appear. To see the Session options, you may have to scroll down the right-side panel.

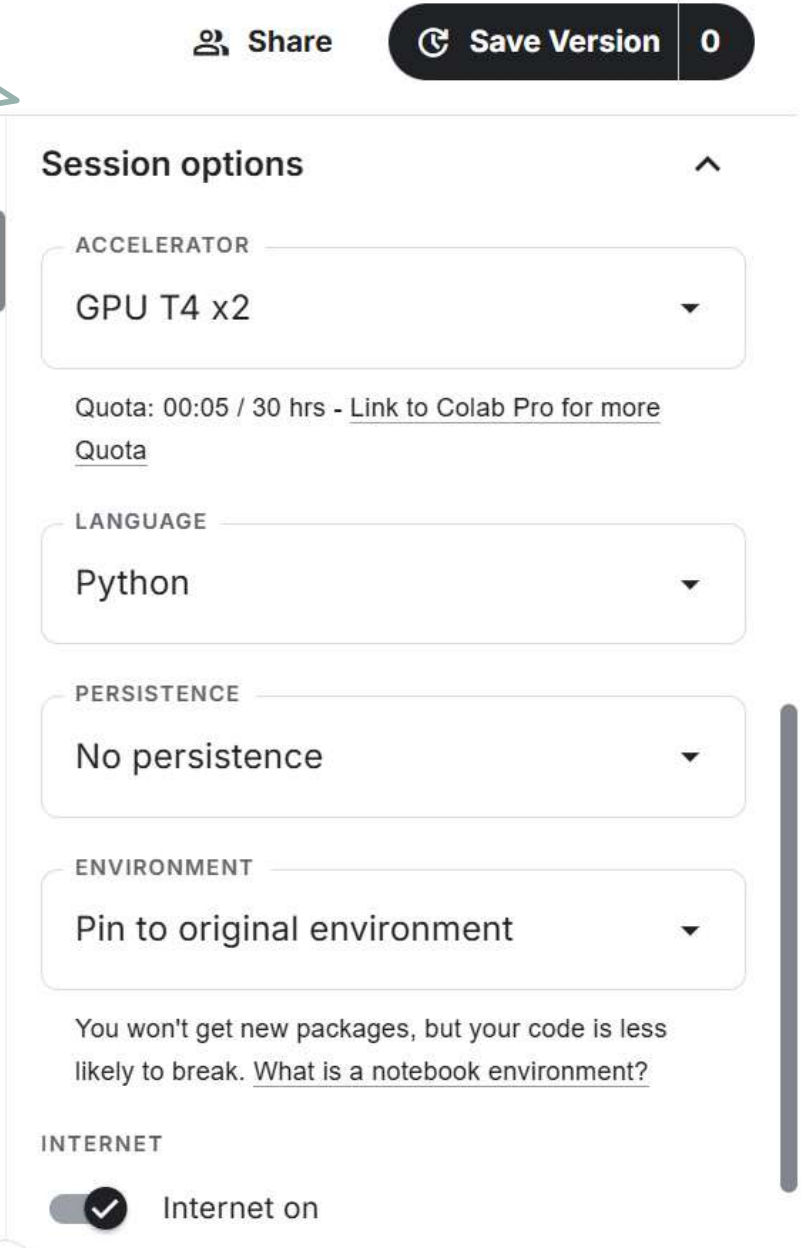
- In the same right-panel, **scroll down to Session options**.
- Go to session options on the right of the screen
- Ensure that you have:
  - GPU T4x2
  - Python
  - No persistence
  - Pin to original environment
  - Internet On

Graphical Processing Unit helps the code to run in parallel which it quicker than Central processing Units. GPUs can run a huge number of simple tasks at the same time.

Pin to original environment means that the software will run with the exact same environment (Kaggle base – such as python version)

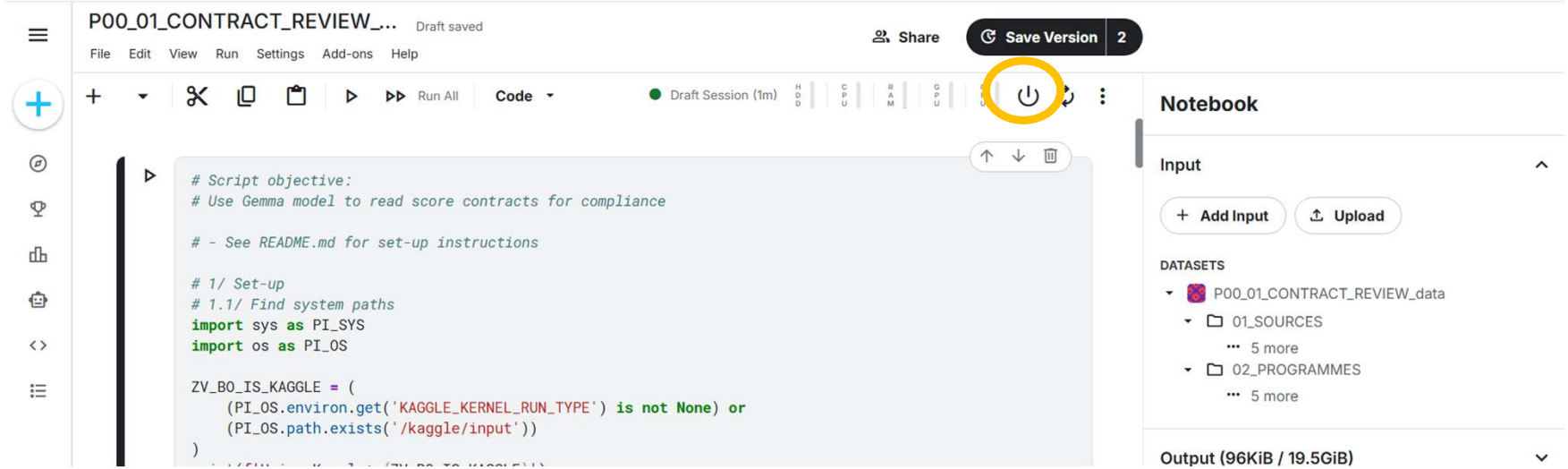
No persistence means that the temporary files and libraries will not be remembered the next time you come to this notebook

We need Internet on, to be able to download the model from HuggingFace.



Kaggle: Turn on the virtual machine

- Now we turn on the machine by pressing the on button:



- When we turn on the machine, Kaggle will ***automatically***:
  - Allocate a VM for you in the cloud
  - Create an isolated container (similar to Docker) in that VM. The container contains:
    - Python
    - Jupyter Notebook
    - Common libraries: NumPy, Pandas, Polars, PyTorch, Tensorflow
    - Optional GPU drivers, if you enabled GPU

Kaggle: Test and run the code

- Once the Kaggle is on, you can optionally change the code, to switch to test mode, by un-commenting the following:

```
# For testing
# ZV_BO_TEST_MODE = True
# ZV_BO_TEST_MODE_WO_LLM = True

# 3/ Test mode - overwrite variables
```

```
ZV_ST_RESULTS_FILE = ZV_DI_VARIABLES.get('ZV

# For testing
ZV_BO_TEST_MODE = True
ZV_BO_TEST_MODE_WO_LLM = True

# 3/ Test mode - overwrite variables
```

- We may want to run the code in test mode first, in order to run it quickly and check that everything is ok in terms of the uploaded files and the AM_VARIABLES, etc.
- Once the code runs fine in test mode, we can comment back the above lines and run it again.
- If we make changes, we can click on More-> Restart and run all cells. If we don't Make changes, we can click on Run All.

What do you think are the advantages of having auditors who know python?

20260610_03_AdvantagesOfKnowingPython?

1. What do yout think are the advantages of having auditors that know how to use python? (Multiple choice)

- [ ] Agility to work with AI models and build APIs so that basic / repetitive tasks can be automated
- [ ] Ability to run any needed analytics to filter large data sets and support additional audit questions
- [ ] Python and data skills enable deeper analysis and stronger investigative work
- [ ] Python skills allow auditors to challenge the responses from auditees by checking against data sets
- [ ] Python lets auditors use AI more effecitvely and move beyond basic chat features
- [ ] Python driven automation increases audit efficiency and reduces reliance on IT

03

Let’s dissect the code!

Automatically install requirements

CONTRACT_REVIEW > 02_PROGRAMMES > Z_SHARED_FUNCTIONS > FC_INSTALL_REQUIREMENTS.py > FC_INSTALL_REQUIREMENTS

```python
from pathlib import Path as PI_PATH
import subprocess as PI_SUBPROCESS
import sys as PI_SYS
import os as PI_OS

def FC_INSTALL_REQUIREMENTS(*,ZVFCI_ST_02PROG_FOLDER):

    ZV_OB_ROOT_FOLDER = PI_PATH(ZVFCI_ST_02PROG_FOLDER)

    # 1/ Find requirements.txt
    ZV_LI_OB_PATH_REQ_FILES = list(
        ZV_OB_ROOT_FOLDER.rglob('requirements.txt')
    )
    if not ZV_LI_OB_PATH_REQ_FILES:
        raise FileNotFoundError(f'No requirements.txt file found under {ZV_OB_ROOT_FOLDER}')

    ZV_OB_PATH_REQUIREMENTS_FILE = ZV_LI_OB_PATH_REQ_FILES[0]

    # 2/ Install python requirements
    PI_SUBPROCESS.run(
        [
            PI_SYS.executable,
            '-m',
            'pip',
            'install',
            '-r',
            str(ZV_OB_PATH_REQUIREMENTS_FILE)
        ],
        check=True
    )
```

We use subprocess to run the install of the libraries mentioned in requirements.txt

Python libraries used in our program

CONTRACT_REVIEW > 02_PROGRAMMES > requirements.txt

```
transformers==4.50.3
huggingface_hub==0.30.2
accelerate==0.34.2
sentencepiece==0.2.0
safetensors==0.4.5
protobuf>=4.25.3,<6
python-docx==1.1.2
PyPDF2==3.0.1
polars==1.40.1
python-dotenv==1.0.1
ipython==8.37.0
pandas==2.2.3
openpyxl==3.1.5
xlsxwriter==3.2.0
```

In requirements we put the versions that are compatible with each other and that are compatible with Python 3.12 (because that is the version of Python that Kaggle uses in its VMs) ... at the time of writing!

Python library	Functionality
transformers	• Download pretrained models from HuggingFace • Load tokenizers, that convert text into numbers • Run text generation: classification, summaries, translations • Fine-tune models based on your dataset • Use models with PyTorch, TensorFlow or Jax
huggingface_hub	• Log in to HuggingFace
accelerate	• Manages hardware: CPU vs GPU selection, multiple GPUs, etc
sentencepiece	• Convert text into smaller pieces – splits out words into pieces also to handle rare words, misspellings, multiple languages, etc
safetensors	• Ensures models load quickly and without malicious code in model weights
protobuf	• Very fast shipping container for data
Python-docx	• Read DOCX documents
PyPDF2	• Read PDF files
polars	• Very fast handling of datasets
python-dotenv	• Importing variables from a file to a dictionary
ipython	• Coding tips if you are using Jupyter Notebooks
pandas	• Handling datasets
openpyxl	• Read and write XLSX
xlsxwriter	• Wrtie new .xlsx files with formatting features

Non-Python libraries used in our program

In order to read .doc files, we need antiword. This is a Linux library. If we are on Linux (not Windows) we can use subprocess to install this Linux library.

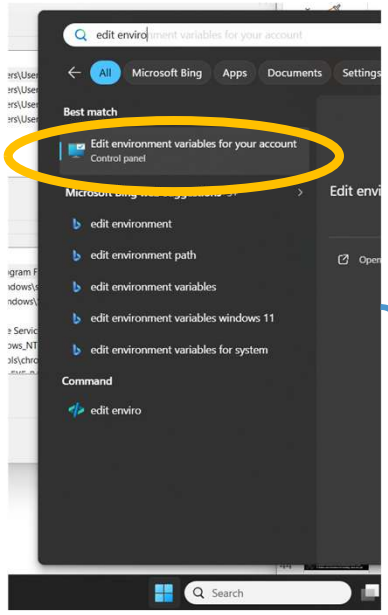
If we are in Windows, the below code will be ignored. For this case, we need to install antiword manually, see next slide. Note: if we are on Kaggle, the below will run, because Kaggle runs our code inside Linux.

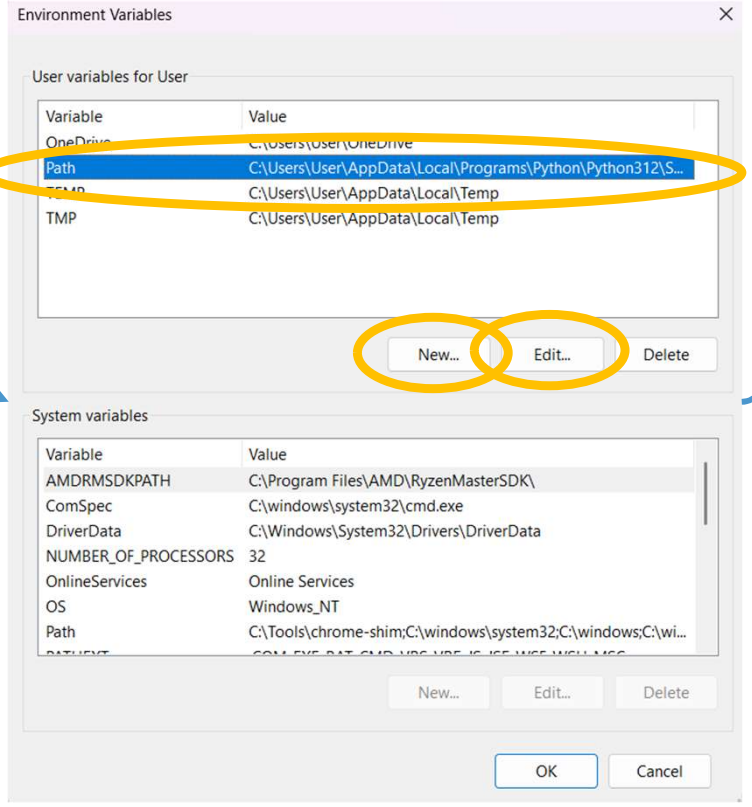
```
if PI_OS.name != 'nt':
    try:
        PI_SUBPROCESS.run(
            ['apt-get', 'install', '-y', 'antiword'],
            check=True
        )
    except Exception as ZV_OB_ERROR:
        raise RuntimeError('antiword was not installed') from ZV_OB_ERROR
else:
    print('Skipping apt-get antiword install on Windows.')
```

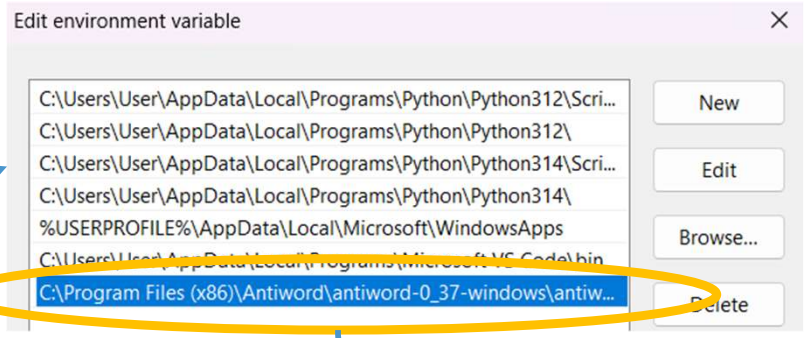
Adding antiword.exe to a Windows environment

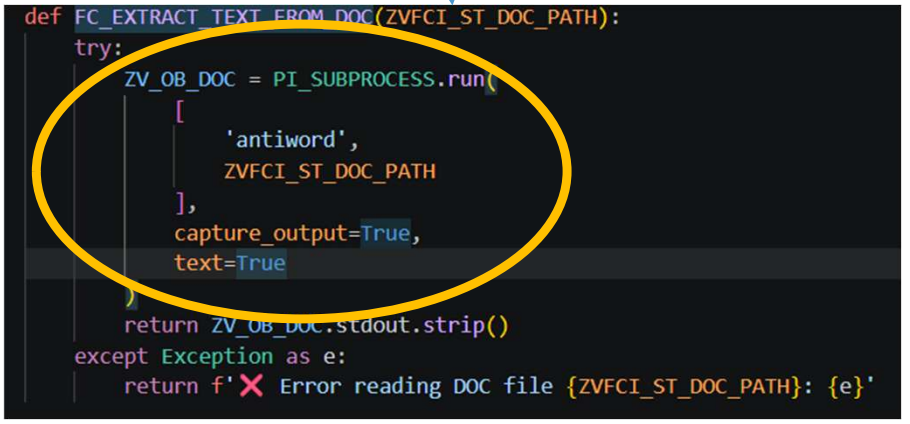
Since antiword is a linux application, we need to manually add it to our Windows path before we can use it.

1. Download antiword from here: https://www.softpedia.com/get/Office-tools/Other-Office-Tools/Antiword.shtml?utm_source=chatgpt.com#download
2. Unzip it in a folder, for example: C:\Program Files (x86)\Antiword
3. Click on Windows and search for Edit environment variables for your account
4. Select the row Path in the box User variables for User
5. With Path selected, under the box User variables for User, click on New
6. Paste the address of the folder that holds antiword.exe
7. You can click on Edit to check it is added
8. Close Visual Studio Code
9. Open Visual Studio Code
10. Now the antiword,.exe should be added to the system path and Python should be able to use it when it runs subprocess.run([‘antiword’,...









```
def FC_EXTRACT_TEXT_FROM_DOC(ZVFCI_ST_DOC_PATH):
    try:
        ZV_OB_DOC = PI_SUBPROCESS.run(
            [
                'antiword',
                ZVFCI_ST_DOC_PATH
            ],
            capture_output=True,
            text=True
        )
        return ZV_OB_DOC.stdout.strip()
    except Exception as e:
        return f'❌ Error reading DOC file {ZVFCI_ST_DOC_PATH}: {e}'
```

1/ Set-up: identify whether or not Kaggle is being used

```
# 1/ Set-up
# 1.1/ Find system paths
import sys as PI_SYS
import os as PI_OS

ZV_BO_IS_KAGGLE = (
    (PI_OS.environ.get('KAGGLE_KERNEL_RUN_TYPE') is not None) or
    (PI_OS.path.exists('/kaggle/input'))
)
print(f'Using Kaggle: {ZV_BO_IS_KAGGLE}')
```

Here, we check whether or not we are using Kaggle. We create a Boolean variable, that we can use throughout the script to choose code based on whether we are in Windows or on Kaggle.

1/ Set-up: Automatically find the address for the root of the project and 02_PROGRAMMES

```
if ZV_BO_IS_KAGGLE:
    ZV_ST_KAGGLE_INPUT_FOLDER = PI_OS.path.join(
        PI_OS.sep,
        'kaggle',
        'input'
    )

    ZV_LI_ST_02PROG_FOLDERS = []

    for ZV_ST_CURRENT_FOLDER, ZV_LI_ST_SUBFOLDERS, ZV_LI_ST_FILES in PI_OS.walk
    (ZV_ST_KAGGLE_INPUT_FOLDER):
        if PI_OS.path.basename(ZV_ST_CURRENT_FOLDER) == '02_PROGRAMMES':
            ZV_LI_ST_02PROG_FOLDERS.append(ZV_ST_CURRENT_FOLDER)

    if not ZV_LI_ST_02PROG_FOLDERS:
        raise FileNotFoundError('No 02_PROGRAMMES folder found under /kaggle/input')

    ZV_ST_02PROG_FOLDER = ZV_LI_ST_02PROG_FOLDERS[0]

    ZV_ST_ROOT_FOLDER = PI_OS.path.dirname(
        ZV_ST_02PROG_FOLDER
    )
else:
    ZV_ST_02PROG_FOLDER = PI_OS.path.dirname(
        PI_OS.path.abspath(__file__)
    )

    ZV_ST_ROOT_FOLDER = PI_OS.path.dirname(
        ZV_ST_02PROG_FOLDER
    )
```

Here, we search for the folder path for the root of the project and for the 02_PROGRAMMES folder for Kaggle, by assuming that we have uploaded our 01_SOURCES and 02_PROGRAMMES to \kaggle\input\

For Windows, we can use __file__ for the name of this .py file and abspath() to know its location. Assuming we put our .py in 02_PROGRAMMES, we can get the directory address with dirname() and then the project root address with dirname() on the folder for 02_PROGRAMMES.

1/ Set-up: Add 02_PROGRAMMES to system path, import functions for install requirements and creating variables

```
PI_SYS.path.append(ZV_ST_02PROG_FOLDER)

# 1.2/ Find and install requirements
from Z_SHARED_FUNCTIONS.FC_INSTALL_REQUIREMENTS import FC_INSTALL_REQUIREMENTS
FC_INSTALL_REQUIREMENTS(ZVFCI_ST_02PROG_FOLDER=ZV_ST_02PROG_FOLDER)

# 1.3/ Find varables and create dictionary
from Z_SHARED_FUNCTIONS.FC_CREATE_DI_VARIABLES import FC_CREATE_DI_VARIABLES
ZV_DI_VARIABLES= FC_CREATE_DI_VARIABLES(ZVFCI_ST_ROOT_FOLDER=ZV_ST_ROOT_FOLDER)
```

Here we use sys.path.append() to add the 02_PROGRAMMES_FOLDER to the system path. After doing that, we can then refer to folders that are inside that folder. For example, Z_SHARED_FUNCTIONS. This allows us to then import our functions.

FC_INSTALL_REQUIREMENTS automatically reads the requirements.txt from the sources folder and installs all the libraries, using the subprocess() function.

```
def FC_INSTALL_REQUIREMENTS(*,ZVFCI_ST_02PROG_FOLDER):

    ZV_OB_02PROG_FOLDER = PI_PATH(ZVFCI_ST_02PROG_FOLDER)

    # 1/ Find requirements.txt
    ZV_LI_OB_PATH_REQ_FILES = list(
        ZV_OB_02PROG_FOLDER.rglob('requirements.txt')
    )
    if not ZV_LI_OB_PATH_REQ_FILES:
        raise FileNotFoundError(f'No requirements.txt file found under {ZV_OB_02PROG_FOLDER}')

    ZV_OB_PATH_REQUIREMENTS_FILE = ZV_LI_OB_PATH_REQ_FILES[0]

    # 2/ Install python requirements
    PI_SUBPROCESS.run(
        [
            PI_SYS.executable,
            '-m',
            'pip',
            'install',
            '-r',
            str(ZV_OB_PATH_REQUIREMENTS_FILE)
        ],
        check=True
    )
```

FC_CREATE_DI_VARIABLES creates a dictionary of variables from .env and AM_VARIABLES

```
def FC_CREATE_DI_VARIABLES(*,ZVFCI_ST_ROOT_FOLDER):

    ZV_OB_ROOT_FOLDER = PI_PATH(ZVFCI_ST_ROOT_FOLDER)

    # 1/ Find .env file
    ZV_LI_OB_PATH_ENV_FILES = list(
        ZV_OB_ROOT_FOLDER.rglob('.env')
    )
    if not ZV_LI_OB_PATH_ENV_FILES:
        raise FileNotFoundError(f'No .env file found under {ZV_OB_ROOT_FOLDER}')

    ZV_OB_PATH_ENV_FILE = ZV_LI_OB_PATH_ENV_FILES[0]

    # 2/ Find AM_VARIABLES.txt
    ZV_LI_OB_PATH_AM_VARIABLE_FILES = list(
        ZV_OB_ROOT_FOLDER.rglob('AM_VARIABLES.txt')
    )
    if not ZV_LI_OB_PATH_AM_VARIABLE_FILES:
        raise FileNotFoundError(f'No AM_VARIABLES.txt file found under {ZV_OB_ROOT_FOLDER}')

    ZV_OB_PATH_AM_VARIABLES_FILE = ZV_LI_OB_PATH_AM_VARIABLE_FILES[0]


    # 6/ Import dotenv module
    from dotenv import dotenv_values as PI_DOTENV_VALUES

    # 7/ Create environment dictionary
    ZV_DI_ENV_VARIABLES = PI_DOTENV_VALUES(ZV_OB_PATH_ENV_FILE)

    # 8/ Create final variables dictionary
    ZV_DI_VARIABLES = PI_DOTENV_VALUES(ZV_OB_PATH_AM_VARIABLES_FILE)
    ZV_DI_VARIABLES.update(ZV_DI_ENV_VARIABLES)

    return ZV_DI_VARIABLES
```

1/ Set-up: Import libraries

```
# 1.3/ Import libraries
from huggingface_hub import login as PI_HUGGINGFACE_HUB_LOGIN
import torch as PI_TORCH
import torch._dynamo
from transformers import AutoTokenizer as PI_AUTOTOKENIZER
from transformers import AutoModelForCausalLM as PI_AUTOMODELFORCAUSALLM
import docx as PI_DOCX
from PyPDF2 import PdfReader as PI_PDFREADER
import subprocess as PI_SUBPROCESS
import time as PI_TIME
import re as PI_RE
import pandas as PI_PANDAS
from IPython.display import display as PI_DISPLAY, HTML
import polars as PI_POLARS

# 1.4/ Import other custom functions
from Z_SHARED_FUNCTIONS.FC_EXPORT import FC_EXPORT_EXCEL_POLARS
```

Once we have installed the libraries, we can then import the libraries.

Once we have imported the libraries, we can then import any other shared functions that might use those libraries, such as FC_EXPORT.

2/ Variables

```
# 2/ Variables
ZV_ST_MODEL_ID = ZV_DI_VARIABLES.get('ZV_ST_MODEL_ID')
ZV_ST_HUGGINGFACE_KEY = ZV_DI_VARIABLES.get('ZV_ST_HUGGINGFACE_KEY')
ZV_ST_SOURCES_FOLDER = PI_OS.path.join(
        ZV_ST_ROOT_FOLDER,
        '01_SOURCES'
    )
if ZV_BO_IS_KAGGLE:
    ZV_ST_RESULTS_FOLDER = PI_OS.path.join(
        PI_OS.sep,
        'kaggle',
        'working'
    )
else:
    ZV_ST_RESULTS_FOLDER = PI_OS.path.join(
        ZV_ST_ROOT_FOLDER,
        '03_RESULTS'
    )


ZV_BO_TEST_MODE = ((ZV_DI_VARIABLES.get('ZV_ST_TEST_MODE')).lower()=='true')
ZV_LI_CATEGORIES = [
    ZV_ST_CATEGORY.strip()
    for ZV_ST_CATEGORY in ZV_DI_VARIABLES.get('ZV_ST_CATEGORIES').split(',')
]
ZV_BO_TEST_MODE_WO_LLM = ((ZV_DI_VARIABLES.get('ZV_ST_TEST_MODE_WO_LLM')).lower()=='true')
ZV_ST_RESULTS_FILE = ZV_DI_VARIABLES.get('ZV_ST_RESULTS_FILE')

# For testing
# ZV_BO_TEST_MODE = True
# ZV_BO_TEST_MODE_WO_LLM = True
```

After having run the function FC_CREATE_DI_VARIABLES(), we can get the variables that we need, from the dictionary.

3/ Login to HuggingFace and download the tokenizer and the LLM model

```python
# 3/ Test mode - overwrite variables
if ZV_BO_TEST_MODE:
    ZV_ST_MODEL_ID = 'sshleifer/tiny-gpt2'
    ZV_LI_CATEGORIES = ['Payment terms']
    ZV_NU_MAX_FILES = 1
    ZV_NU_MAX_CHARS = 2000
else:
    ZV_ST_MODEL_ID = ZV_DI_VARIABLES.get('ZV_ST_MODEL_ID')
    ZV_NU_MAX_FILES = None
    ZV_NU_MAX_CHARS = None

# 3/ Obtain the LLM model objects
print('Start of model download:')
print(f'Model: {ZV_ST_MODEL_ID}')
print(f'Test mode: {ZV_BO_TEST_MODE}')
print(f'LLM disabled: {ZV_BO_TEST_MODE_WO_LLM}')
print(f'Categories: {len(ZV_LI_CATEGORIES)}')

if ZV_BO_TEST_MODE_WO_LLM:
    ZV_OB_TOKENIZER = None
    ZV_OB_LLM_MODEL = None

else:
    # 3.1/ Login to HuggingFace
    PI_HUGGINGFACE_HUB_LOGIN(ZV_ST_HUGGINGFACE_KEY)

    # 3.2/ Create a tokenizer object for the model
    ZV_OB_TOKENIZER = PI_AUTOTOKENIZER.from_pretrained(ZV_ST_MODEL_ID)

    if ZV_OB_TOKENIZER.pad_token is None:
        ZV_OB_TOKENIZER.pad_token = ZV_OB_TOKENIZER.eos_token

    # 3.3/ Create an LLM object for the model (float 16 for reduced memory)
    ZV_OB_LLM_MODEL = PI_AUTOMODELFORCAUSALLM.from_pretrained(
        ZV_ST_MODEL_ID,
        device_map='auto',
        torch_dtype=PI_TORCH.float16
    )

    # 3.4/ Disable memory-efficient and flash attention (optional for debugging or
    compatibility)
    PI_TORCH.backends.cuda.enable_mem_efficient_sdp(True)
    PI_TORCH.backends.cuda.enable_flash_sdp(True)
```

If we are not in test mode, we login to HuggingFace and get the tokenizer and the model.

The tokenizer is an object that will be used to break up the text into pieces of text and then into numbers.

The model that is downloaded depends on the name of the model that is set in AM_VARIABLES.txt. For this project we set it to be google/gemma-3-1b-it, which is a light-weight Neural Network model.

The model is set to use float16, which makes it not so heavy, compared to if we use a model with more detailed numbers (more decimals) to represent the words.

device_map='auto' means that the model will automatically choose the best device to run on.

PI_TORCH is used to optimize speed and memory usage.

5/ Extract data from the files, depending on their extension

```python
# 5.1/ Function: Extract text from DOCX files ---
def FC_EXTRACT_TEXT_FROM_DOCX(ZVFCI_ST_DOCX_PATH):
    ZV_ST_TEXT = ''
    ZV_OB_DOC = PI_DOCX.Document(ZVFCI_ST_DOCX_PATH)
    for ZV_OB_PARAGRAPH in ZV_OB_DOC.paragraphs:
        ZV_ST_TEXT += ZV_OB_PARAGRAPH.text + '\n'
    return ZV_ST_TEXT.strip()

# 5.2/ Function: Extract text from PDF files ---
def FC_EXTRACT_TEXT_FROM_PDF(ZVFCI_ST_PDF_PATH):
    ZV_ST_TEXT = ''
    with open(ZVFCI_ST_PDF_PATH, 'rb') as ZV_OB_FILE:
        ZV_OB_READER = PI_PDFREADER(ZV_OB_FILE)
        for ZV_OB_PAGE in ZV_OB_READER.pages:
            if ZV_OB_PAGE.extract_text():
                ZV_ST_TEXT += ZV_OB_PAGE.extract_text() + '\n'
    return ZV_ST_TEXT.strip()


# 5.3/ Function: Extract text from DOC (97-2003) using antiword ---
def FC_EXTRACT_TEXT_FROM_DOC(ZVFCI_ST_DOC_PATH):
    try:
        ZV_OB_DOC = PI_SUBPROCESS.run(
            [
                'antiword',
                ZVFCI_ST_DOC_PATH
            ],
            capture_output=True,
            text=True
        )
        return ZV_OB_DOC.stdout.strip()
    except Exception as e:
        return f'❌ Error reading DOC file {ZVFCI_ST_DOC_PATH}: {e}'
```

Here we read the file into a simple text variable.

The way that we read the file, depends on the file extension.

Here, we can see that for .doc files, we use the antiword Linux executable that we added to the Windows path.

6/ Create a prompt with the file text, create a chat, iwth the tokenizer

```python
def FC_GENERATE_RESPONSE(ZVFCI_ST_CONTENT, ZVFCI_LI_CATEGORIES):

    ZV_DI_OUTPUTS = {}
    ZV_TI_START = PI_TIME.time()

    for ZV_ST_CATEGORY in ZVFCI_LI_CATEGORIES:

        ZV_ST_PROMPT = (
            f"Extract only the exact information about '{ZV_ST_CATEGORY}' from the following
            contract. "
            "Do not explain, summarize, or rephrase. If not found, answer 'None'.
            \n\nContract:\n"
            + ZVFCI_ST_CONTENT
        )
```

Here we are creating a PROMPT.

The PROMPT will contain some instructions.

The instructions include the category.

They also include the text from one file.

We then use the Tokenizer object to create the chat template from our prompt.

A chat template is a format that converts a conversation into the exact text structure that the LLM model expects. Different models: different formats. For example,

We might write:
User: What is a contract?
Assistant: A contract is...

However, the model might expect the chat to be written differently.

```python
ZV_LI_DI_CHAT = [
    {'role': 'user', 'content': ZV_ST_PROMPT}
]

if ZV_OB_TOKENIZER.chat_template is None:
    ZV_ST_CHAT_TEMPLATE = ZV_ST_PROMPT
else:
    ZV_ST_CHAT_TEMPLATE = (
        ZV_OB_TOKENIZER
        .apply_chat_template(
            ZV_LI_DI_CHAT,
            tokenize=False,
            add_generation_prompt=True
        )
    )
```

6/ Tokenize the prompt into numbers, send it to the LLM model in CPU or in GPU

```
    else:
        ZV_LI_LI_INPUT_TOKEN_IDS = (
            ZV_OB_TOKENIZER
            .encode(
                ZV_ST_CHAT_TEMPLATE,
                add_special_tokens=False,
                return_tensors='pt'
            )
            .to(ZV_OB_LLM_MODEL.device)
        )
```

Using the chat template, that contains the prompt, which contains our text from the file, we ***tokenize*** the message.

This means that all of the words are converted into complex numbers. Some words are split into word pieces.

pt means ‘return the result as a PyTorch tensor’ which is kind of like a safe, fast, datatype.

The tensor object is moved to the same device where the pre-trained LLM model is already stored. This could be CPU or GPU, depending on the settings we chose earlier.

Hello contract

tensor([[15496,4321]])

6/ Get back the response as numbers from the LLM, decode it back to English with the tokenizer

```
    else:
        ZV_LI_LI_OUTPUT_TOKEN_IDS = (
            ZV_OB_LLM_MODEL
            .generate(
                input_ids=ZV_LI_LI_INPUT_TOKEN_IDS,
                do_sample=True,
                temperature=0.3,
                top_p=0.9,
                max_new_tokens=512
            )
        )

    ZV_ST_OUTPUT = (
        ZV_OB_TOKENIZER
        .decode(
            ZV_LI_LI_OUTPUT_TOKEN_IDS[0][ZV_LI_LI_INPUT_TOKEN_IDS.shape[-1]:],
            skip_special_tokens=True
        )
        .strip()
    )

    ZV_DI_OUTPUTS[ZV_ST_CATEGORY] = ZV_ST_OUTPUT

print('✅ Response generated in', round(PI_TIME.time() - ZV_TI_START, 2), 'seconds')

return ZV_DI_OUTPUTS
```

The LLM receives the encoded message.

If we set do_sample, it will choose tokens based on probability and not always the most likely.

temperature =0.3 means it will not choose very randomly, so it will behave in a stable way, less creatively.

top-p=0.9 means “nucleus sampling”: only choose from the tokens in top 90%, avoiding unlikely tokens

Max new tokens, means the maximum number of tokens in the response. 512 tokens is about 350 English words.

The outputs are added to a dictionary, which is returned. The LLM gives a response in numbers. The Tokenizer object decodes the response back to normal words.

Hello! How can I help you with your contract?

tensor([[15496, 703, 460, 314, 1037, 345, 351, 534, 7035, 30]])

7/ Clean-up the response

```python
def FC_CLEAN_RESPONSE(ZVFCI_DI_OUTPUTS, ZFCI_ST_FILENAME):

    ZV_LI_CATEGORIES = []
    ZV_LI_SENTENCES = []

    # Loop through all extracted information
    for ZV_ST_CATEGORY, ZV_ST_STRING in ZVFCI_DI_OUTPUTS.items():

        # Split text into sentences
        ZV_LI_ST_SENTENCES = PI_RE.split(r'(?<!\w\.\w.)(?<![A-Z][a-z]\.)(?<=\.|\?|:)\s', ZV_ST_STRING)

        # Remove irrelevant filler sentences
        ZV_LI_ST_SENTENCES = [
            ZV_ST_SENTENCE for ZV_ST_SENTENCE in ZV_LI_ST_SENTENCES
            if 'cannot provide'.lower() not in ZV_ST_SENTENCE.lower() and
               'cannot extract'.lower() not in ZV_ST_SENTENCE.lower() and
               'the text does not contain'.lower() not in ZV_ST_SENTENCE.lower() and
               'sure, the information related'.lower() not in ZV_ST_SENTENCE.lower() and
               'sure, here is information'.lower() not in ZV_ST_SENTENCE.lower() and
               'sure, here is the information'.lower() not in ZV_ST_SENTENCE.lower()
        ]

        # Join filtered sentences
        ZV_ST_SENTENCES = ' '.join(ZV_LI_ST_SENTENCES)
        if ZV_ST_SENTENCES.strip() == '':
            ZV_ST_SENTENCES = 'None'

        # Clean up extra tokens and formatting
        ZV_ST_SENTENCES = ZV_ST_SENTENCES.replace('<eos>', '').replace('\n', ' ').strip()

        ZV_LI_CATEGORIES.append(ZV_ST_CATEGORY)
        ZV_LI_SENTENCES.append(ZV_ST_SENTENCES)

    # ✅ Create a clean, organized DataFrame
    ZV_DF = PI_PANDAS.DataFrame({
        'ZF_ST_FILENAME': ZFCI_ST_FILENAME,          # <-- Auto-filled real file name
        'ZF_ST_CATEGORY': ZV_LI_CATEGORIES,
        'ZF_ST_SENTENCES': ZV_LI_SENTENCES
    })

    # Display the result
    return ZV_DF
```

Convert LLM response into a table:

- Each output dictionary is for one file and contains LLM responses for each category (see ZV_ST_CATEGORIES in AM_VARIABLES)
- Each LLM response/ category combination is:
  - Split into sentences
  - Unwanted sentences are removed
  - Special words, such as <eos> (“End of sequence”) are removed
  - The sentences are all joined together
  - The string and the category are saved to two lists: one for sentences and one for the category.
- The resulting lists are entered into a dictionary that also has the file name.
- The dictionary is converted to a DataFrame: which will have the fields: ZF_ST_FILENAME| ZF_ST_CATEGORY| ZF_ST_SENTENCES.
- The DataFrame is returned.

8/ Run all of the above: starting with reading the files

```python
# Initialize an empty list to store all response DataFrames
ZV_LI_DF = []
ZV_LI_ALLOWED_EXTENSIONS = ['docx', 'pdf', 'doc']

# Iterate through each file in the dataset directory
for ZV_NU_FILE_INDEX, ZV_OB_FILE in enumerate(PI_OS.listdir(ZV_ST_SOURCES_FOLDER)):

    ZV_ST_EXTENSION = ZV_OB_FILE.split('.')[-1].lower()

    if ZV_ST_EXTENSION not in ZV_LI_ALLOWED_EXTENSIONS:
        print(f'⚠️ Skipping non-contract file: {ZV_OB_FILE}')
        continue

    if ZV_NU_MAX_FILES is not None and len(ZV_LI_DF) >= ZV_NU_MAX_FILES:
        break

    ZV_OB_FILEPATH = PI_OS.path.join(ZV_ST_SOURCES_FOLDER, ZV_OB_FILE)
    ZV_ST_FILENAME = PI_OS.path.basename(ZV_OB_FILE).split('.')[0]

    print(f'Processing file: {ZV_ST_FILENAME}')

    # --- Detect file type and extract text accordingly ---
    if ZV_OB_FILEPATH.split('.')[-1] == 'docx':
        print('📘 DOCX file found.')
        ZV_ST_CONTRACT_TEXT = FC_EXTRACT_TEXT_FROM_DOCX(ZV_OB_FILEPATH)

    elif ZV_OB_FILEPATH.split('.')[-1] == 'pdf':
        print('📕 PDF file found.')
        ZV_ST_CONTRACT_TEXT = FC_EXTRACT_TEXT_FROM_PDF(ZV_OB_FILEPATH)

    elif ZV_OB_FILEPATH.split('.')[-1] == 'doc':
        print('📗 DOC (97-2003) file found.')
        ZV_ST_CONTRACT_TEXT = FC_EXTRACT_TEXT_FROM_DOC(ZV_OB_FILEPATH)

    else:
        print('⚠️ File type not recognized. Skipping this file.')
        continue
```

For each file

- Find the extension
- Get the text for the file using the FC_EXTRACT_TEXT_FROM_DOCX or FC_EXTRACT_TEXT_FROM_PDF or FC_EXTRACT_TEXT_FROM_DOC
- ...

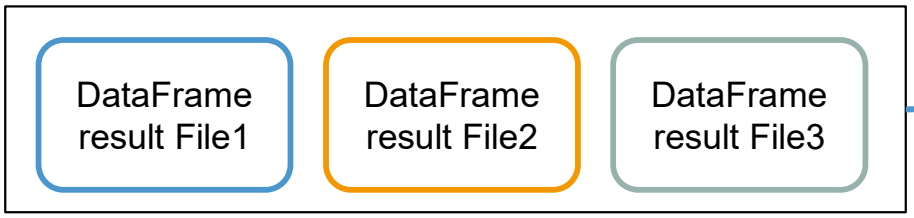
8/ Run all of the above: .. and then getting responses, cleaning responses and.. grouping into one DataFrame

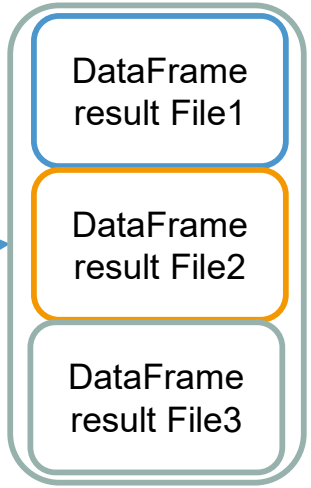
```
        ZV_DI_RESPONSE = FC_GENERATE_RESPONSE(
            ZV_ST_CONTRACT_TEXT,-
            ZV_LI_CATEGORIES
        )
    print('✅ Response generation complete!')
    # --- Step 3: Clean and structure the model response ---
    print('🧹 Cleaning model response ...')
    ZV_DF_OUTPUTS = FC_CLEAN_RESPONSE(ZV_DI_RESPONSE, ZV_ST_FILENAME)
    print('✅ Response cleaning complete!')
    print('='*50)

    # Append the cleaned response to the main list
    ZV_LI_DF.append(ZV_DF_OUTPUTS)

# ========================
# STEP 8 : Combine, Format, and Display Final Results
# ========================

# 🧩 Configure pandas display options
PI_PANDAS.set_option('display.max_rows', 50)          # Limit rows di
PI_PANDAS.set_option('display.max_columns', 50)       # Limit columns
PI_PANDAS.set_option('display.max_colwidth', 200)     # Allow wider t

# 🧠 Combine all response DataFrames into a single table
ZV_DF_ALL_OUTPUTS = PI_PANDAS.concat(ZV_LI_DF, ignore_index=True)
```

For each file ....

- Run the FC_GENERATE_RESPONSE() function using the text that was obtained and the list of categories
- Run the FC_CLEAN_RESPONSE() function using the resulting dictionary from FC_GENERATE_RESPONSE and the file name
- Put the resulting DataFrame from FC_CLEAN_RESPONSE into ZV_LI_DF.

Once all of the files have been processed, we have a list of dataframes: a list where each item is a DataFrame that contains the fields: file name| Category| Response.





We finally convert our list of DataFrames into one table that contains all of the results.

Result

Formatted full table preview (scroll horizontally if needed):

Contract name	Contract Information	Details
master-service-agreement-template-09	Contract Date	None.
master-service-agreement-template-09	Effective Date	Effective Date: July 1, 2021.
master-service-agreement-template-09	Renewal Term	None.
master-service-agreement-template-09	Exit clause incl. notice period	None.
master-service-agreement-template-09	Contract Parties	None.
master-service-agreement-template-09	Documents Retention Period	None.
master-service-agreement-template-09	Audit Clause	None.
master-service-agreement-template-09	Audit Frequency	None.
master-service-agreement-template-09	Audit Duration	None.
master-service-agreement-template-09	Contract Fees	None.

Now we can see which contracts are compliant – and which ones are missing key information.

- ✓ Done based on meaning – not based on word search
- ✓ Done for PDFs or for Word documents
- ✓ Done for multiple contracts
- ✓ Takes about 10 minutes for 4 contracts
- ✓ We could leave it running overnight for 100s contracts and then check the result in the morning!

Since the above took too much CPU/ GPU-> ChatGPT!

- Downloading and using an NLP model can be costly in terms of CPU/GPU
- So you might not be able to run it directly on your laptop
- In this case, you would rather run it using a ChatGPT API
- Or an API to your own in-house Open AI instance
- The principle is similar
- In fact, ChatGPT, open AI, askSage, etc. … they are all just pre-trained models, like Gemma!
- It just depends if you want to subscribe to a paid service, or if you want to use free, Open-Source models that you can perhaps tweak more precisely – from HuggingFace
- If you only need to score contracts based on whether or not a category is present, then a free Gemma model might be a good option, especially if you have thousands of contracts and don't want to spend money on tokens, or send data outside your VPN.


Questions?