SAP Data Audit Master Class

Python cheat sheet
All the python you need for all the 300 must have data audit tests

80% of what you need to know for python for the 300 must-have data analytics tests in the 300Framework Masterclass

Legend

Tip

Note: all of the - that you see in the code should be the short - (minus) and not the long – used for writing.

Links will be shown in blue font

```
Code in the CMD box
```

```
Code in Visual Studio Code PowerShell terminal
```

```
Code in a python script
```

```
Code in other file types: .toml, .txt, .env, .json
```

Set-up

Set-up

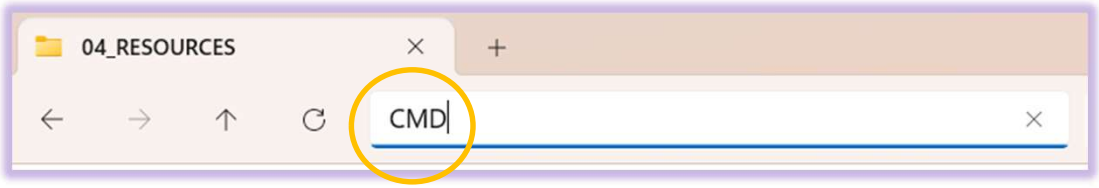
To start using python, you need to do the following 4 steps:

1/ Install python

To get set-up with python you need to first download python and install it on your computer:

s://www.python.org/downloads

Then, open a command window, by clicking on the Windows search bar and typing CMD, followed by enter:



A small CMD box will open. To check if you have correctly installed python, type the following in the CMD box:

```
python --version
```

2/ Install a code editor

We will use the free code editor: Visual Studio Code. Download and install it from here:

s://code.visualstudio.com/Download

3/ Install python extension in Visual Studio Code

To enable Visual Studio Code to colour your code based on python: Open Visual Studio Code on your computer and click on Extensions. In the search bar of extensions, type python. Then select to install Python.

4/ Allow yourself to run programs (not required on all PCs)

In Visual Studio Code, go to the menu View and choose Terminal. Check that the terminal that opens is a Powershell terminal: the prompt should start with PS. If it is not a Powershell terminal, choose Powershell from the drop-down next to the + on the top-right of the terminal.

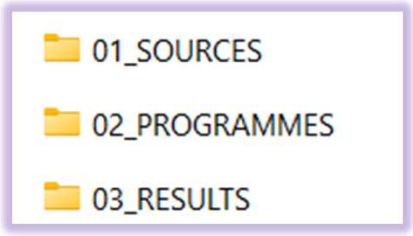
Allow yourself to execute programs, by typing the following in the PS terminal: (ensure the - are - and not –)

```
Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
```

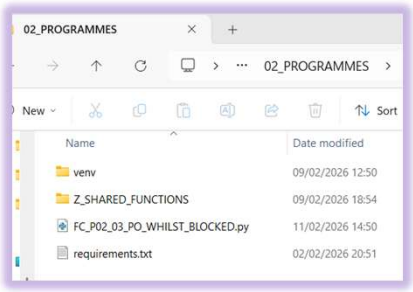
Project organization

Project folder organization

When creating projects, always use the following project folder structure, to avoid confusing source files with results files or program files. This ensures that you maintain a clear audit trail for your work.



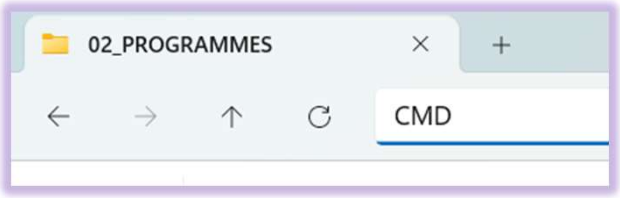
Inside 02_PROGRAMMES, you should have Z_SHARED_FUNCTIONS, where scripts that are used by other scripts should go.



Open Visual Studio Code in the correct location

In Windows, navigate to inside the 02_PROGRAMMES folder. Click in the Windows address bar.

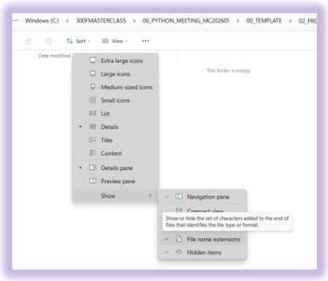
Then type CMD to open a command window in that location.



Visual Studio Code will open in the correct location for your project.

Create a toml file for your project

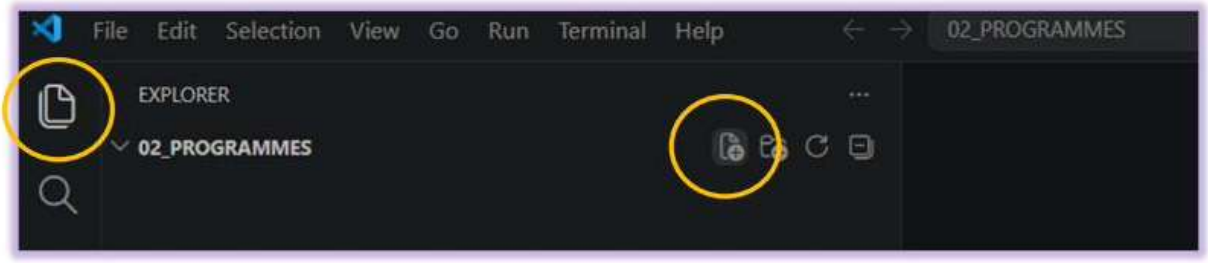
Show file extensions



When using python, it is useful to know the types of the files that we are analysing. Ensure that you can see the file extensions in Windows. In any Windows folder, go to View -> Show and ensure that File name and extensions is ticked.

Create a .toml file

In Visual Studio Code, click on the Explorer icon on the top left and then click on new file. Create a file called: pyproject.toml.



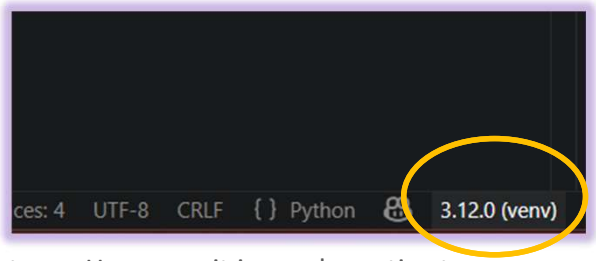
Open the file by clicking on it and then enter the following:

```
[project]
requires-python = ">=3.11,<3.13"
```

[project] indicates the project section of the .toml file. Requires-python is a variable name.

The above example is for if the python version that you will put in your virtual environment is 3.10 (see next page). You may change the version numbers in the above example, if you intend to put a different python version in your virtual environment.

To see the version of the python.exe that is found in your virtual environment, after creating and activating the virtual environment, as mentioned on the next page, you can type python --version in the terminal, or you can open a script and check the version number in the bottom right of the screen.



The .toml file is for information purposes only or for use by external systems. However, it is good practice to put the python version in the .toml file so that anyone using your project later – on a different machine - knows which version to use when they create the virtual environment.

For example, if you create your project in version 3.10, but later, someone tries to run it with version 3.20, it is possible that the program will bug due to some incompatibilities between the python libraries in 3.10 and the other libraries that you will import (as documented in requirements.txt – see further on), or even incompatibilities due to plain or built-in python functions, classes or methods that are no longer found in later python versions.

TOML stands for Tom´s Obvious Minimal Language. Tom Preston-Werner (one of the co-founders of GitHub in 2008).

Create a virtual environment for your project

Open a PowerShell terminal

With Visual Studio Code open in 02_PROGRAMMES, go to the menu View and then Terminal.

A terminal will appear at the bottom of the screen. Check that the prompt-line in the terminal window starts with PS.

If it starts with PS, it means that it is a PowerShell terminal. If it is not a PowerShell terminal, click on the arrow next to the + at the top-right of the terminal. Choose PowerShell.

Create and activate virtual environment

All python projects should have their own virtual environment. We will then put all python libraries, used in our project, inside this virtual environment. This ensures that they can always run, even if the version of python libraries changes.

In the Visual Studio Code PowerShell terminal, type the following to create a virtual environment in the folder venv: (note: here we are assuming using the python veresion 3.10, but you might want to use a later version as mentioned in your .toml file (see previous page).

```
py -3.12 -m venv venv
```

The above means:

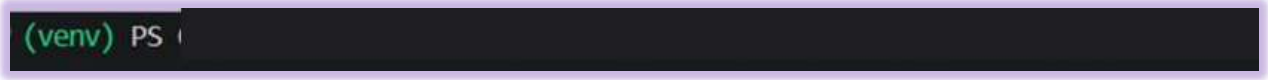
- python: use the python launcher
- -3.12: choose version 3.12
- -m venv: run the virtual environment module
- .venv: create the folder venv in the current folder: current folder indicated by .

The - means “this is an instruction for the command”. Be careful not to put the long dash rather than the shorter -.

Once the venv folder has appeared in the 02_PROGRAMMES folder, we can activate it by running the activate.ps1 file. Type the following to activate the virtual environment. (Note the extension .ps1, because we are in a PowerShell terminal, so we are running the .ps1 file).

```
.\venv\Scripts\Activate.ps1
```

If you have successfully activated your virtual environment, you will see a green (venv) text in front of the prompt line:



You should also click/hover on the python version number at the bottom-right of Visual Studio Code. When you hover over venv() it will show you the address of the venv, which should be the address of the venv inside your project. If not: click and choose the correct venv by using the browse feature. ***Both*** **mentions of venv should be showing the address of the venv in your project:**

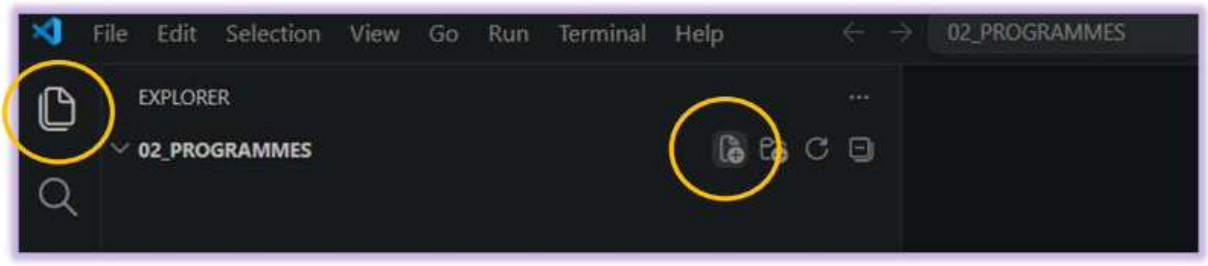
Create a requirements.txt file

Create requirements.txt

All python projects should have a requirements.txt file. This file will hold the versions of the python libraries that we use in our project. If we come back to our project in six months time, it may not work with different library versions. This is why we need to ensure that we record the library versions in the requirements file.

Inside Visual Studio Code, click on the Explorer button on the top-left of the screen, to open the Explorer side panel on the left.

At the 02_PROGRAMMES folder level, click on the button to add a file:

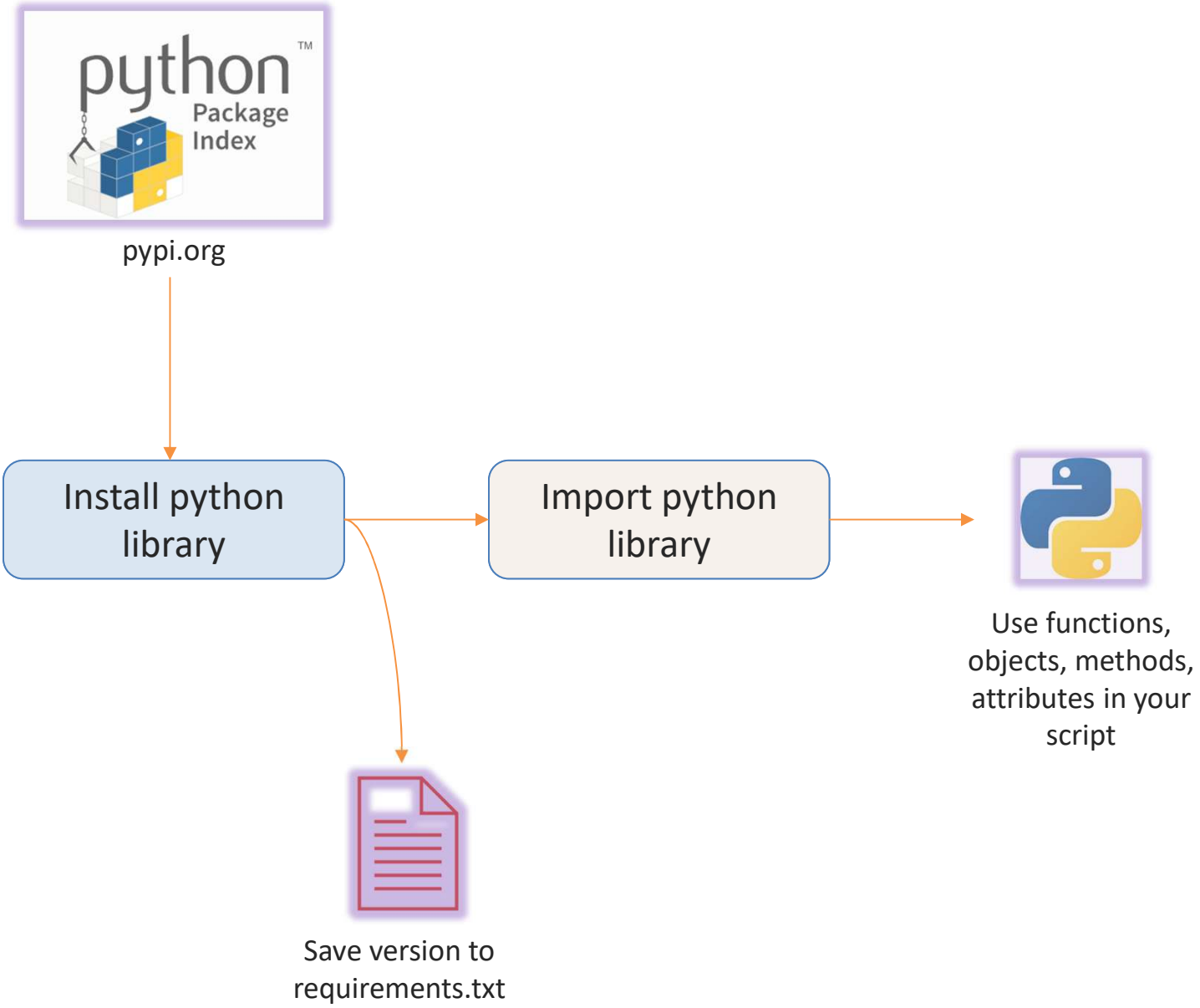


Name the file requirements.txt. As you add python libraries to your project, you will add the mention of these libraries and their versions to the requirements.txt file, as mentioned on the next two pages.

Install and import a library

The main advantage of python, apart from the fact that it is free and that it is the main language of Artificial Intelligence, is that it comes with endless libraries, that can enable you to do pretty much anything, with a minimal amount of code.

To benefit from all the functionality offered by the 100s 1000s libraries, we need to understand how to install, import and use python libraries.



See the end of this document for examples of python libraries that are used in the 300Framework.

Install and import a python library

Install a python library

The python Polars library enables us to run analysis on data sets very fast. For example, filter, sort, group_by(), join, etc. We can find all of the documentation on this library here: **https://docs.pola.rs/**

In the Visual Studio Code PowerShell terminal, that is activated for your virtual environment, type the following code to download the latest version of the Polars library into your virtual environment:

```
pip install polars
```

The Polars library will be installed to the virtual environment and you can find it in the venv folder.

Record in requirements.txt for later use

Whenever we install a library, we will record it in requirements.txt. To find the version of the Polars library that you need to record, type the following in the Visual Studio Code PowerShell terminal:

```
pip show polars
```

Find where it says version: ... then record the version in requirements.txt as follows:
(<libraryName>==<versionNumber>)

```
polars==1.35.0
```

To re-install libraries to your venv, in case you set-up your project on a different computer, type:

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

Import a library

To use a python library in your script, you need to import it. For example:

```
import polars
```

Hide secrets

Hide secrets in an environment file and load during run-time

We often want to be able to do APIs (Application Program Interface) to different systems. Typically, when doing an API, we will need to pass a secret key. It is important that we do not put the secret key in our python code file. Instead, we should put the key in an environment file.

An environment file should always be called .env. A .env file is typically ignored when we put our code into GitHub or other places. Furthermore, .env files are typically hidden files that you cannot see in Windows, unless you specifically show hidden files.

The .env will contain information such as:

```
ZV_ST_CHATGPT_KEY=<your-secret-key-goes-here>
```

Quotation marks are not required. There should be no space between the parameter name, the = and the value: <parameterName>=<value>.

We use the library module load_dotenv from the library dotenv to load the variables from the .env file into the process environment (the temporary memory that is used whilst the program is running, also known as the RAM).

We use the parameter override=True to mean: if the variable already exists, override it.

Once the variable is in memory, we use the getenv function of the os library to access it.

We name the variable with the same name in our script, for ease of script maintenance.

```
import os as PI_OS
from dotenv import load_dotenv as PI_LOAD_DOTENV

PI_LOAD_DOTENV(override=True)

ZV_ST_CHATGPT_KEY = PI_OS.getenv('ZV_ST_CHATGPT_KEY')
```

In this way, our secret key remains in our .env file but is not shown in our script.

if you are using GitHub, then you should always have a gitignore file in the project that mentions .env as a file that should not be sent to GitHub.

Use external variables – don´t hard-code

Import and use variables from an external text file

Variables may be collected from users as inputs to a front-end dashboard. Variables may also be documented in a text file. For example, the date that the data was downloaded from the SAP system.

We put variables that are not entered by the users in the front-end dashboard, in a text file called AM_VARIABLES.txt. This text file should be located in the 01_SOURCES folder.

In the AM_VARIABLES.txt file, all variables should be written without quotation marks, except for lists of text, for example, the variable Language:

```
AnalysisStartDate=20220101
AnalysisEndDate=20231231
Language='E, EN'
MaximumRAM=20
MaximumCPU=5
```

The following python script is FC_CONFIG_LOADER.py. It puts the values from the variabes script into a dictionary.

```
from pathlib import Path as PI_PATH
from dotenv import load_dotenv as PI_LOAD_DOTENV
from dotenv import dotenv_values as PI_DOTENV_VALUES
import os as PI_OS

PI_LOAD_DOTENV(override=True)

ZV_ST_VARIABLES_FILE = (
    PI_PATH(PI_OS.getenv('ZV_ST_PROJECT_ROOT'))
    / '01_SOURCES'
    / 'AM_VARIABLES.txt'
)
ZV_DI_VARIABLES = PI_DOTENV_VALUES(ZV_ST_VARIABLES_FILE)
```

We put the FC_CONFIG_LOADER.py into the Z_SHARED_FUNCTIONS folder, and then we call it in order to create the dictionary of variables in local memory, so that it can be used by our script:

```
from Z_SHARED_FUNCTIONS.FC_CONFIG_LOADER import ZV_DI_VARIABLES
```

In the above snippet, instead of importing a function, we have imported an object.

Now, we can access the values in the dictionary. For example, we might want to get the analysis start date as a string or the maximum RAM as an integer:

```
AnalysisStartDate = ZV_DI_VARIABLES.get('AnalysisStartDate')
MaximumRam = int(ZV_DI_VARIABLES.get('MaximumRAM'))
```

Naming conventions & rules

Naming conventions & rules (1/3)

When writing python code, always use the following naming conventions, to ensure your code is quick to read and understand. The below rules also help us to avoid classic errors.

Example	Rule
`import polars as PI_POLARS`	Imported libraries: PI_<LIBRARY:NAME>: (PI: PythonLibrary Import) Use Polars wherever possible, as it runs faster than Pandas.
`.read_csv()`	Code from python libraries in lower case
`UPPER`	Anything that we make in upper case
`FC_IMPORT()`	Functions that we make: prefix FC_
`CL_BANK_ACCOUNT()`	Classes that we make: prefix CL_
`ME_DEPOSIT()`	Methods within classes that we make: prefix ME_

`FC_IMPORT (ZVFCI_ST_SOURCE_FILE,`
`ZVFCI_ST_FOLDER,`
`ZVFCI_ST_DELIMITER):`
	Input variables imported into our functions:
	prefix ZVFCI_ (Z: custom, V: variable, FC:
	function, I: Input)

`ZV_ST_FILE_PATH`	Variable that we create that is of type string: prefix ZV_ST_ (Z: custom, V: Variable, ST: string)
`ZV_DT_START_DATE`	Variable that we create that is of type date: (Z: custom, V: variable, DT: date)
`ZV_NU_THRESHOLD`	Variable that we create that is of type numeric (Z: custom, V: variable, NU: numeric)
`ZV_LI_LANGUAGES`	Variable that we create that is of type list: prefix ZV_LI: (Z: custom, V: variable, LI: list)
`ZV_DI_APPROVAL_LIMITS`	Variable that we create that is of type dictionary: prefix ZV_DI (Z: custom, V: variable, DI: dictionary)
`ZV_DF_JE_LINES`	Variable that we create that is of type dataframe (for example, when we want to work on a dataframe section in a group_by): (Z: custom, V: variable, DF: dataframe)
`ZV_SET_TCODES`	Variable that we create that is of type set (Z: custom, V: variable, SET: set)
`ZV_SE_BKPFTCODE_TSTCTTTEXT`	Variable that we create that is of type series (column): prefix: ZV_SE (Z: custom, V: variable, SE: series)
`ZV_OB_FILE`	Variable that we create that is another type of object – not mentioned above: prefix ZV_OB (OB: object)

Naming conventions & rules (2/3)

Example	Rule
A_<SAPTableName>	SAP table that we imported into our program
<SAPTableName>_<SAPFieldName>	Name of SAP fields that we imported to our program. For example, BSEG_DMBTR.
ZF_<SAPTableName>_<SAPFieldName>_<Action/Description>	Name of field that we created based on a SAP field, for example, ZF_BSEG_DMBTR_USD. Note: generally speaking, if we sum a field in a group_by(), we do not change the name of the field summed.
<SAPTableName>_<SAPFieldName>_<SAPTableFieldAddedTo>	Name of field that we added to another table. For example, if we added the transaction code description field (TSTCT_TTEXT) to the general ledger header (BKPF) table, then we would write TSTCT_TTEXT_BKPF. This enables us to not confuse the TSTCT_TTEXT field that is added to other tables, such as MKPF.
ZF_<SAPTableName><SAPFieldName>_<SAPFieldName>	Name of the field that we create when we concatenate the description with the code. For example, if we concatenate the transaction code description (TSTCT_TTEXT) with the transaction code (BKPF_TCODE), then we would write ZF_BKPFTCODE_TTEXT
ZF_ST_<Description>	For example, ZF_ST_JE_TYPE. If we create a string field to describe the journal entry type: such as manual / normal.
<ScriptNumber>_<99>_TT_DF_<SAPTable>_<Action/Description> for example: B01_01_TT_DF_BSEGBKPF_FILTER	Temporary dataframes (tables) that we make: prefix: script number that made the dataframe, _<99>_DF_TT (<99>: chronological order of creation, DF: dataframe, TT: Temporary Table)
<ScriptNumber>_<99>_IT_DF_<SAPTable>_<Description> for example: B01_12_IT_DF_BSEGBKPF_CUBE	Dataframes that we will use later: prefix: script number that made the dataframe, _<99>_DF_IT (<99>: chronological order of creation, DF: dataframe, IT: Interpreted Table)
<ScriptNumber>_<99>_RT_DF_<SAPTable>_<Description> for example: B01_13_RT_DF_BSEGBKPF_TOTALS	Dataframes that we will later present in the 300Framework dashboard and that show us information, such as totals per supplier.
<ScriptNumber>_<99>_XT_DF_<SAPTable>_<Description> for example: B01_13_XT_DF_BSAIK_DUP_PAY	Dataframes that we will later present in the 300Framework dashboard and that show us exceptions to internal control rules, such as duplicate payments.

Naming conventions & rules (3/3)

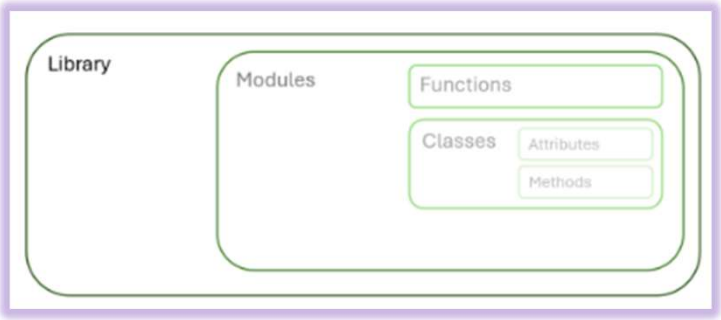
Example	Rule
B01_01_TT_BSEG_COUNT = (A_BSEG .group_by(BSEG_BUKRS, BSEG_GJAHR, BSEG_BELNR) .agg(PI_POLARS.col('BSEG_BELNR') .n_unique() .alias('ZF_BSEG_BELNR_COUNT')))	Write code vertically for fast half-screen review. We often split our screen in half to compare two sets of code, or compare code to something else. For example, to compare code from ChatGPT or Claude to our code-base. Therefore, for quick review, we write all of our code vertically. Chain with . for continued steps. Use the Polars library, wherever possible, for fast execution.
# 1/ Import	All steps should have brief comments. Comments should be chronologically numbered.
Obsolete code snippets	Obsolete code must be deleted. This is to ensure that we don't waste time reviewing it and so that we do not confuse AI agents.
FC_<ActionOrServiceDescription> in Z_SHARED_FUNCTIONS – no repetition	Code must not be repeated. Repeated functions, such as import text file, export to Excel, export to SQL database, must go in functions, found in separate python scripts. All such functions must be put in a separate python script in Z_SHARED_FUNCTIONS. This ensures that updates can be done in one central location. For example, if you want to improve the way you import text files, you want to make sure that you only have to modify the import script one time.
Technical names under presentation level, friendly names on the presentation level.	Data that is provided to end-user presentations(dashboards, etc) must be provided with technical names. Friendly names must be shown to the user on the presentation level. (on the dashboard). This is to ensure that dashboards are easy to read, but that information relating to the technical origin of the data remains up until the last level, for quick maintenance and review purposes.
pyproject.toml	All projects must have a pyproject.toml that indicates the python version that was used when the virtual environment was created.
.venv	All projects must have a virtual environment folder, in which the python.exe inside the .venv/Scripts is the same as that mentioned in the .toml.
requirements.txt	All projects must have a requirements.txt file that lists all of the libraries that need to be installed for the scripts in the project to work.
.env	All projects must have a .env file for storing secrets. Keys and passwords should never be included in the python script itself.
no hard-coding	Variables should never be hard-coded. All variables must come from a user-input or a parameters file. This is to avoid future errors due to values, such as start date, or end date, etc. that are forgotten inside a script.
Short scripts – one function per script	Scripts should be written with one purpose only and they should be as short as possible. This is to ensure that they are easy to maintain, but also to ensure that the RAM is released between each operation, to prevent RAM overload.

Python components

How is python organized?

Python is organized as follows:

- Libraries
    - Modules / sub-modules (namespaces)
        - Functions
        - Classes
            - Attributes
            - Methods



- Functions: do actions.
- Classes: create objects.
- Objects: have attributes and methods
- Methods do actions on objects or attributes of objects in the same class.

We can download libraries and use their functions and classes.

We can also make our own functions and classes.

Functions

Functions do an action.

A function can ally take objects as parameters.

A function can ally return objects as results.

Example: Function that does not take or return any objects:

```
def FC_HELLO_WORLD():
    print("hello world")
```

Example: Function that takes an object:

```
def FC_WELCOME(ZVFCI_ST_NAME):
    print(f"Hi {ZVFCI_ST_NAME}, Welcome to the 300Framework Masterclass")
```

This function takes the object type string variable as an input.

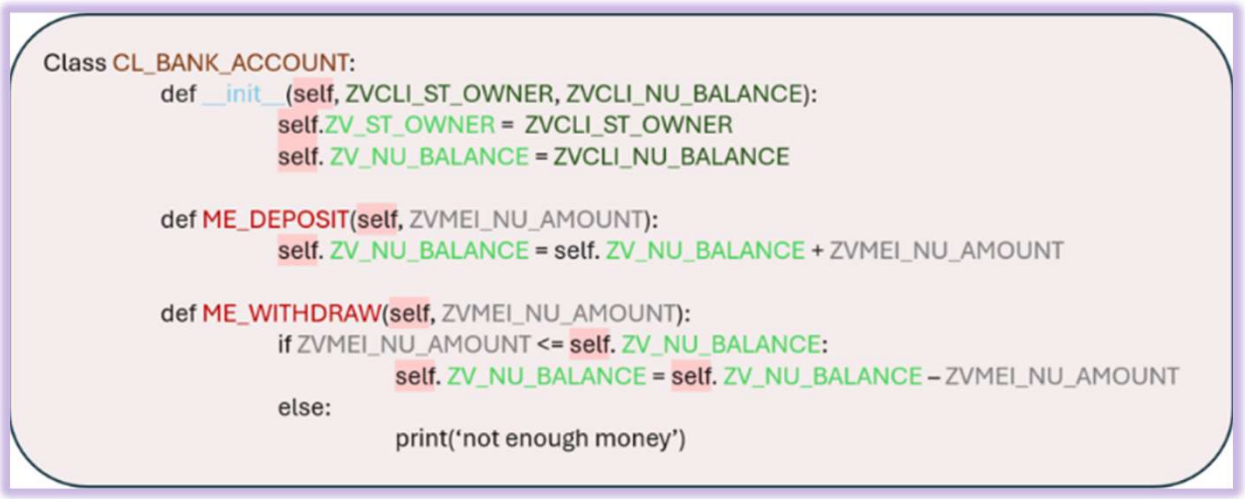
Example: Function that takes and returns objects:

```
def FC_BALANCE(ZVFCI_DF_BSID):
    ZV_DF_BSID_KUNNR_TOT = (
        ZVFCI_DF_BSID
        .group_by(
            BSID_KUNNR
        )
        .agg(
            PI_POLARS.col('ZF_BSID_DMBTR_USD_SIGNED')
            .sum()
        )
    )
    return ZV_DF_BSID_KUNNR_TOT
```

This function takes a dataframe object (customer outstanding items) in its signature and returns a dataframe object (total outstanding per customer)

Classes

Below, is an example of a custom class, that we can create, in order to understand what a class is and how to use it.



Using this class, we can create a bank account object: ZV_OB_ACCOUNT. This object is an instance of the CL_BANK_ACCOUNT class.

We can use this bank account object to deposit and withdraw, using the methods of the CL_BANK_ACCOUNT class.

```
ZV_OB_ACCOUNT = CL_BANK_ACCOUNT('Claire', 1000)
ZV_OB_ACCOUNT.ME_DEPOSIT(250)
ZV_OB_ACCOUNT.ME_WITHDRAW(100)
```

Notice how we replace self with ZV_OB_ACCOUNT.

Objects

In python, everything is an object. Dataframes, columns, variables, lists and dictionaries are all examples of objects. The functions that we can use on an object, depend on the object class definition.

For example, a dataframe can be a Polars dataframe or a Pandas dataframe. If the dataframe is a Polars dataframe, then we can only use Polars functions on it and not Pandas functions.

Dataframe

A dataframe in python is almost the same as a table. A dataframe has columns (also referred to in python literature as Series) and rows.

Column

We use PI_POLARS.col() to access a column in a Polars dataframe. Once accessed, functions, methods and operations can be done on all items of the column. For example, PI_POLARS.col('ZF_ST_MY_COL').str.strip_chars() removes all leading/ trailing spaces from every item in ZF_ST_MY_COL.

Variable

A variable is one value. We can think of a variable as a box with one value inside. When working on variables, we can use normal python functions, rather than Polars functions.

List

In python, a list is written with [] brackets. A list can contain any number of items. Each item can be of a different object class (dataframe, variable, list, dictionary). Items in the list can be accessed based on the index (position in the list). The first position in a list has the index 0.

A list is written as [<object>, <object>, <object>], where object can be of any type (text, numeric, list, dictionary, …). Objects can be of different types within the same list. The same object can occur more than once.

```
ZV_LI_MY_LIST = [object, object, object, object]
```

Each position in the list has an index, starting at 0:

- 1st object: index 0
- 2nd object: index 1
- 3rd object: index 2
- 4th object: index 3

Objects

Dictionary

In python, a dictionary is written with {} brackets. A dictionary can contain any number of items. Each item can be of a different object class (dataframe, variable, list, dictionary, …). Items in the dictionary can be accessed based on the key, which is the name of the item.

A dictionary is written as {"<keyName>": <object>}, where object can be of any type (text, numeric, list, dictionary,…). Objects can be of different types within the same dictionary. We typically use nested dictionaries.

```
ZV_DI_MY_DICTIONARY = {
    'key': {
        'key': {
            'key': myObject
        }
    }
}
```

Group_by()

When we use group_by(), we obtain a group_by() object. A group_by() object is a dataframe that has been divided up into sub-dataframes.

```
ZV_OB_GROUPBY = (
   ZV_DF
   .group_by(
      [
        'Material',
        'Month',
      ]
   )
)
```

Material	Month	Posting date	Input date	Input time	Document	Item	Quantity	Value
Bolts	Jan 2025	1 Jan 2025	1 Jan 2025	00:01:30	00001234	0001	10	200
Bolts	Jan 2025	2 Jan 2025	2 Jan 2025	00:05:20	00001235	0001	3	60
Bolts	Jan 2025	3 Jan 2025	3 Jan 2025	00:16:10	00001236	0001	2	400
Bolts	Jan 2025	25 Jan 2025	25 Jan 2025	00:01:30	00001237	0001	14	280
Bolts	Feb 2025	1 Feb 2025	1 Jan 2025	00:01:30	00001238	0001	5	1000
Bolts	Feb 2025	2 Feb 2025	1 Jan 2025	00:05:20	00001241	0001	6	1200
Bolts	Feb 2025	3 Feb 2025	1 Jan 2025	00:16:10	00001242	0001	17	3400
Bolts	Feb 2025	25 Feb 2025	1 Jan 2025	00:01:30	00001256	0001	2	400
Bolts	Mar 2025	1 Mar 2025	1 Mar 2025	00:01:30	00001281	0001	21	4200
Bolts	Mar 2025	28 Mar 2025	28 Mar 2025	00:05:20	00001290	0001	32	6400

As soon as we apply a method on the group_by() object, it is transformed back into a dataframe. Typically, we apply the method agg() to a group_by() object and use other methods on columns inside agg():

```
   .group_by(…)
   .agg(
     [
      PI_POLARS.col('..').sum().alias(''),
      PI_POLARS.col('..').mean().alias(''),
      PI_POLARS.col('..').first().alias(''),
      PI_POLARS.col('..').n_unique().alias('')
     ]
   )
```

In the above code, you can see that we can use the Polars col() function to bring up a column and then we can apply chained methods on that column. This gives us a lot of flexibility.

Custom

As mentioned above – a class is used to define an object. You can make your own class and so your own objects. Your objects can have their own customized attributes and methods. An attribute is also an object: for example, a string variable that holds a name.

Datatypes

Datatypes determine the function, method, operands

The functions, methods or operands (+,-,*,&,|) that we can use on an object, depend on the object and also on the type.

If our object is a variable or a column, it can also be of type, string, date, numeric or Boolean.

If our object is a list, the items in the list can be of type string, date, numeric or Boolean.

Sometimes, we need to convert between different types. The function that we use to do the conversion depends on the class and the type.

For example, the function to convert a variable of type string to a variable of type numeric is different to the function that we use to convert a Polars column of type string to numeric or a Pandas column of type string to numeric.

Python syntax

Indent

In python, code has to be written on the correct tab indent from the left. For example, everything after if… : has to be indented by one tab, if it is to be done only when the if condition is true.

Parenthesis ()

Parenthesis are used to allow code to go on a separate line – so that we can write code vertically – as mentioned in our naming convention. Parenthesis are also used on function names. They hold the signature, which is the list of variables that can be fed to the function.

:

The colon is used after the name/ expression of **functions**, **for**, **if**. For example, if … : for … : FC_MY_FUNCTION(…):

Square brackets []

Square brackets are used to hold items in a list. Lists are typically seen when a function requires a list as an input parameter. For example, group_by([]), which takes a list of column names.

Curly braces {}

Curly braces are used to hold items in a dictionary. Dictionaries are typically seen when a function requires a dictionary as an input parameter. For example, rename({}), which takes a dictionary that has the current column names as the keys and the new column names as the values.

.

. means "apply my function or method to the object or attribute before the dot". Or "give me the attribute of an object": <object>.<attribute>. Or “access the function that is in the library or name-space of …

```
ZV_DF = (
    ZV_DF
    .with_columns(
        PI_POLARS.col('BSEG_HKONT')
        .str.strip_chars()
    )
)
```

This means: my dataframe object is equal to my dataframe object, to which you should apply the with_columns() method. Use the col() function from the Polars library to get the column BSEG_HKONT, to which you should apply the method strip_chars(), which is part of the str() namespace. Note: col() is a function, whereas as with_columns, str and strip_chars() are methods. Functions that are not built-in (those coming from imported libraries) have the library name infront of the dot. Methods have the object name infront of the .

&, |, ==, !=, ~

Means: **AND**, **OR**, **is equal to**, **is not equal to**, **NOT**

=

Means assign. For example, ZV_ST_MY_NAME = 'Claire'

Code snippets

Dataframe: print to view / access columns

Print top of a file to the terminal

We often want to print to terminal to view a dataframe during processing: We use f'{}' to call code, or expose the content of variables, whilst printing:

```
print(f'ZV_DF: {ZV_DF.head(10)}')
```

The above code prints the first 10 rows of the dataframe to the terminal.

Obtain the list of columns in a dataframe

We often want to know the list of columns in a dataframe, for example to check that we have all of the columns that we need for our audit test:

```
ZV_LI_COLS = ZV_DF.columns
```

In the above code, columns is an attribute of the object dataframe. ZV_LI_COLS is a list of strings, which are the names of the columns.

The syntax <object>.<attribute> enables you to access attributes inside your object.

Because we are accessing the attribute and not calling a method, we do not need paranthesis. So we do not write ZV_DF.columns().

Dataframe: access columns

Process all columns in a dataframe

We may want to process all columns in a dataframe. For example, add the prefix of the SAP table name:

```
for ZV_ST_COL_NAME in ZV_DF.columns:
    ZV_DF = (
        ZV_DF
        .rename(
          {
            ZV_ST_COL_NAME: f'{ZVFCI_ST_PREFIX}_{ZV_ST_COL_NAME}'
          }
        )
    )
```

Process all columns in a dataframe, if found in list

We may want to process all columns in a dataframe, only if they are found in a list. For example, convert some of the columns to numeric datatype:

```
for ZV_ST_COL_NAME in ZVFCI_LI_NUM_COLS:
    ZV_DF = (
        ZV_DF
        .with_columns(
            [
                PI_POLARS.col(ZV_ST_COL_NAME)
                .cast(PI_POLARS.Float64, strict=False)
                .alias(ZV_ST_COL_NAME)
            ]
        )
    )
```

Dataframe: access item

Obtain a value from a dataframe

We often want to get a value from a dataframe. For example, we might want to get the latest date found. We can use select().max() and item():

```
ZV_ST_BSAID_BUDAT_LATEST = (
    ZV_DF_BSAID
    .select(
        PI_POLARS.col('BSAID_BUDAT')
        .max()
    )
    .item()
)
```

Dataframe: filter based on value/ variable

Filter on a hard-coded value

We often want to filter a dataframe to not include rows with blank text or zero values:

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .filter(
        (PI_POLARS.col('EKPO_MATNR') != '') &
        (PI_POLARS.col('EKPO_NETWR') != 0)
    )
)
```

Filter on a variable

We often want to filter a dataframe based on a variable. The variable could be an information that is entered into a parameters file, or to the dashboard, by an end-user.

For example, in the below code, we filter on the year of the start and end date that is passed to our program.

Note: that we use [:4]. Here : means slice. Nothing before the slice means start from the beginning. 4 means up to the 4th character.

Note: for variables we can use [:4] to get the first four digits. However, for columns (series), we need to use the Polars method from the str namespace: ZF_ST_MYCOLUMN.str.slice(0,4).

```
ZV_DF_LFC1 = (
    ZV_DF_LFC1
    .filter(
        (PI_POLARS.col('LFC1_GJAHR') >= ZV_ST_START_DATE[:4]) &
        (PI_POLARS.col('LFC1_GJAHR') <= ZV_ST_END_DATE[:4])
    )
)
```

Dataframe: filter based on list

Found in a list

We often want to know if our general ledger account is found in a list. For example, we might have a list of cash accounts and we want to filter our general ledger to see if it contains those accounts:

```
ZV_LI_BSEG_HKONT_CASH = ['55012345', '550123333']

ZV_DF_BSEG = (
    ZV_DF_BSEG
    .filter(
        PI_POLARS.col('BSEG_HKONT')
        .is_in(
            ZV_LI_BSEG_HKONT_CASH
        )
    )
)
```

Not found in a list

Sometimes, we want to filter on records that are not found in a list. For example, we may want to obtain all suppliers that are not intercompany or personnel. We can use ~, for not.

```
ZV_LI_KTOKK_INTERCO = ['INT', 'GROUP', 'PERS']

ZV_DF_LFA1 = (
    ZV_DF_LFA1
    .filter(
        ~(
            PI_POLARS.col('LFA1_KTOKK')
            .is_in(
                ZV_LI_KTOKK_INTERCO
            )
        )
    )
)
```

Dataframe: group and agg()

Group rows – to calculate sum, mean, etc.

We often want to do operations on groups of rows. For example, the average price per material, or the total value per material. This is also another way of making sure a key is unique before a join.

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .group_by(
        'EKPO_MATNR'
    )
    .agg(
        [
            (
                PI_POLARS.col('ZF_EKPO_NETPR_USD') *
                PI_POLARS.col('EKPO_PEINH') *
                (
                    PI_POLARS.col('EKPO_BPUMN') /
                    PI_POLARS.col('EKPO_BPUMZ')
                )
            )
            .mean()
            .alias('ZF_EKPO_NETPR_USD_MEAN'),

            PI_POLARS.col('ZF_EKPO_NETWR_USD')
            .sum()
        ]
    )
)
```

In the code above, we can see that group_by() is a method that works on a Polars dataframe to return a grouped Polars dataframe. Whereas, agg() is a Polars method that works on a grouped Polars dataframe.

The group_by method takes a string representing a column name, or a list of strings for more than one column name.

The agg() method takes a Polars expression or a list of Polars expressions.

Dataframe: group and agg()

Group rows – to list out words

We often want to list out the words in a string column. For example, we often want to obtain the list of accounts on debit and credit for our journal entries:

```
ZV_DF_BSEG = (
    ZV_DF_BSEG_ACC_SUM
    .sort(
        [
            'BSEG_BUKRS',
            'BSEG_GJAHR',
            'BSEG_BELNR',
            'BSEG_SHKZG',
            'ZF_BSEG_DMBTR_SUM',
        ],
        descending=[False, False, False, False, True]   # <- force numeric desc on the sum
    )
    .group_by(
        [
            'BSEG_BUKRS',
            'BSEG_GJAHR',
            'BSEG_BELNR',
        ],
        maintain_order=True
    )
    .agg(
        [
            (
                PI_POLARS.col('BSEG_HKONT')
                .filter(PI_POLARS.col('BSEG_SHKZG') == 'S')
                .cast(PI_POLARS.Utf8)
                .implode()
                .list.slice(0, 5)
                .list.join('_')
                .alias('ZF_BSEG_HKONT_DEBIT')
            ),
            (
                PI_POLARS.col('BSEG_HKONT')
                .filter(PI_POLARS.col('BSEG_SHKZG') == 'H')
                .cast(PI_POLARS.Utf8)
                .implode()
                .list.slice(0, 5)
                .list.join('_')
                .alias('ZF_BSEG_HKONT_CREDIT')
            )
        ]
    )
)
```

In the above code, we list out the first five accounts on debit, in descending order of the value, creating the column ZF_BSEG_HKONT_DEBIT. We also list out the first ten accounts on credit, in descending order of the value, creating the column ZF_BSEG_HKONT_CREDIT.

Dataframe: sort/ sort([])

Sort a dataframe – one column: sort()

We can sort an entire dataframe. For example, we might want to sort the material movements on input date, in order to see the total value in stock over time: (below we assume that the date is in text format YYYYMMDD so that it will sort properly):

```
ZV_DF_MSEGMKPF = (
    ZV_DF_MSEGMKPF
    .sort(
       'MKPF_CPUDT'
    )
)
```

Sort a dataframe – many columns: sort(by = [])

If we have many fields that we are sorting by, we use the key word by=[]. We can also sort descending, if we want to see the postings in reverse order.

In the below code we include the time-stamp. We also include the document number and item, in order for the result to be "deterministic", meaning always the same.

```
ZV_DF_MSEGMKPF = (
    ZV_DF_MSEGMKPF
    .sort(
        by = [
            'MSEG_BUKRS',
            'MSEG_MATNR',
            'MKPF_CPUDT',
            'MKPF_CPUTM',
            'MSEG_MBLNR',
            'MSEG_ZEILE'
        ],
        descending = [False, False, True, True, True, True]
    )
)
```

Note: the key words **by =** and **descending =** are al *keyword arguments* (also known as **kwargs**). We write them for clarity.

Dataframe: group/ agg: sort

Sort within an aggregation – same field: sort()

If we do not want to keep all records and only want to keep the last date per material, we can use a group by on the material number with a sort to obtain the last date.

```
ZV_DF_MSEGMKPF = (
    ZV_DF_MSEGMKPF
    .group_by(
        [
            'MSEG_BUKRS',
            'MSEG_MATNR'
        ]
    )
    .agg(
        PI_POLARS.col('MKPF_CPUDT')
        .sort(descending=True)
        .first()
        .alias('ZF_MKPF_CPUDT_LATEST')
    )
)
```

Sort within an aggregation – different field: sort_by()

If we also want to keep the value for the last movement, per material, we could sort by a date, time for the value. To be deterministic, we could also include the document number and item.

```
ZV_DF_MSEGMKPF = (
    ZV_DF_MSEGMKPF
    .group_by(
        [
            'MSEG_BUKRS',
            'MSEG_MATNR'
        ]
    )
    .agg(
        PI_POLARS.col('ZF_MSEG_DMBTR_SIGNED')
        .sort_by(
            [
              'MKPF_CPUDT',
              'MKPF_CPUTM',
              'MSEG_MBLNR',
              'MSEG_ZEILE'
            ],
            descending = [True, True, True, True]
        )
        .first()
        .alias('ZF_MSEG_DMBTR_SIGNED_LATEST')
    )
)
```

Dataframe: group/ agg: maintain_order

Maintain_order for a group_by()

If our table was already sorted, we can use the keyword argumente (kwarg) **maintain_order** to keep the order in the **group_by():**

```
ZV_DF_MSEGMKPF = (
    ZV_DF_MSEGMKPF
    .group_by(
        [
            'MSEG_BUKRS',
            'MSEG_MATNR'
        ],
        maintain_order=True
    )
    .agg(
        PI_POLARS.col('MKPF_CPUDT')
        .first()
        .alias('ZF_MKPF_CPUDT_LATEST')
    )
)
```

Dataframe: join: inner/semi/ anti

Join: inner – add columns from another table and filter

We often want to join data together. For example, add header information to detail information, only keep rows that match:

```
ZV_DF_EKPOEKKO = (
    ZV_DF_EKPO
    .join(
        ZV_DF_EKKO,
        left_on  = ['EKPO_EBELN'],
        right_on = ['EKKO_EBELN'],
        how = 'inner'
    )
)
```

Join: semi – keep rows found in another table

We often want to limit one table on another table. For example, only obtain journal entries created by the payment program and don't keep any columns from the payment program:

```
ZV_DF_BSEGBKPF = (
    ZV_DF_BSEGBKPF
    .join(
        ZV_DF_REGUH,
        left_on  = ['BSEG_BUKRS','BSEG_BELNR','BKPF_BUDAT'],
        right_on = ['REGUH_ZBUKR','REGUH_VBLNR','REGUH_ZALDT'],
        how = 'semi'
    )
)
```

Join: anti – remove rows found in another table

We often want to ignore rows if they are in another table. For example, purchase orders without change document (no approval record).

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .join(
        ZV_DF_CDPOS,
        left_on  = ['EKPO_EBELN'],
        right_on = ['CDHDR_OBJECTID'],
        how = 'anti'
    )
)
```

Dataframe: join: left/ full

Join: left – add columns from another table, keep all left

We often want to add information from another table, but not remove any rows from the current table. For example, add the material description, if a description exists:

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .join(
        ZV_DF_MAKT,
        left_on  = ['EKPO_MATNR'],
        right_on = ['MAKT_MATNR'],
        how = 'left'
    )
)
```

Join: full – keep all rows from both tables

We often want to keep all rows from both tables, so that we can find in the next step, any rows from either table that did not match, and also keep all of our values for both data sets. For example, when comparing the total per account in the general ledger, to the total per account in the trial balance:

```
ZV_DF_BSEG_HKONT_TOT = (
    ZV_DF_BSEG_HKONT_TOT
    .join(
        ZV_DF_GLT0_SAKNR_TOT,
        left_on  = ['BSEG_BUKRS','BSEG_GJAHR','BSEG_HKONT'],
        right_on = ['GLT0_BUKRS','GLT0_RYEAR','GLT0_RACCT'],
        how = 'full'
    )
)
```

Dataframe: join: cross

Join: cross – multiply one data frame by another

We often want to multiply one dataframe by another dataframe. For example, if we want to have one row for every day/ time in a window, when looking for mass sales within a particular period.

```
ZV_DF_OFFSETS = PI_POLARS.DataFrame(
    {
        'ZF_NU_OFFSET': list(range(ZV_NU_OFFSET))
    }
)

ZF_VBAPVBAK = (
    ZF_VBAPVBAK
    .with_columns(
        PI_POLARS.col('VBAK_ERDAT')
        .str.strptime(PI_POLARS.Date, '%Y%m%d', strict=False)
        .alias('ZF_VBAK_ERDAT_DT')
    )
    .join(
        ZV_DF_OFFSETS,
        how='cross'
    )
    .with_columns(
        (
            PI_POLARS.col('ZF_VBAK_ERDAT_DT') -
            PI_POLARS.col('ZF_NU_OFFSET')
        )
        .dt.total_days()
        .alias('ZF_VBAK_ERDAT_DT_OFFSET')
    )
)
```

The above code multiplies all lines in the sales order table, for each integer from 0 up to ZV_NU_DAYS_WINDOW.

The code then calculates an offset date, which is the sales order date minus the number of days in the offset.

This new column can then be used in a group_by() to see if there were sales orders within the same time period that add-up to a total sales order value or volume above a particular threshold.

Dataframe: generate a date range

One row per day between two dates

We often need one row per day, for example to left-join a daily FX rate to our postings. PI_POLARS.date_range() produces a series of dates that can be used as a dataframe column. eager=True returns the values immediately instead of as a lazy expression. The values are returned in the form of a Polars series, which can then be included in the dictionary expression for the function DataFrame().

```
ZV_SE_DATES = PI_POLARS.date_range(
    start=ZV_DT_BEG_DATE,
    end=ZV_DT_END_DATE,
    interval='1d',
    eager=True
)

ZV_DF = PI_POLARS.DataFrame(
    {
      'ZF_DT_DAY': ZV_SE_DATES
    }
)
```

Dataframe: unique

Ensure that the key is unique

Before joining two tables, we often want to make sure that the key in the right-hand table is unique. Otherwise, the rows in the left-hand table will be duplicated (except in the case of a semi join). For example, get only one description per material, before adding the description to the list of material movements.

```
ZV_DF_MSEG = (
   ZV_DF_MSEG
   .join(
      ZV_DF_MAKT = (
        ZV_DF_MAKT
        .unique(
            subset=['MAKT_MATNR']
        )
      ),
      left_on = 'MSEG_MATNR',
      right_on = 'MAKT_MATNR',
      how = 'left'
   )
)
```

Note: in the above code, you would also want to make sure that the description is filtered on English language.

Dataframe: pipe

Pipe a dataframe through a shared function

We often want to apply a function from Z_SHARED_FUNCTIONS to a dataframe, while keeping the dot-chain style. Use the Polars method .pipe() to pass the Polars dataframe or expression - to the left of the .pipe() - as the first argument of the function, along with any extra arguments.

```
ZV_DF = (
    ZV_DF
    .pipe(
        FC_MERGE_WITH_NULL_HANDLING,
        ZV_DF_OFAC,
        'ZF_KY_JN_OFAC_LFA1'
    )
)
```

In the above example, our custom function FC_MERGE_WITH_NULL_HANDLING takes three variables: ZV_DF, ZV_DF_OFAC and ' ZF_KY_JN_OFAC_LFA1'.

Dataframe: iterate rows

Loop through each row as a dictionary

We rarely need to iterate rows, because Polars is much faster with vectorised operations. But for row-level logic that cannot be vectorised (for example, expanding a SAP authorization range into individual transaction codes), we use .iter_rows(named=True) to get each row as a dictionary.

```
for ZV_DI_ROW in ZV_DF.iter_rows(named=True):
    ZV_ST_USER = ZV_DI_ROW['USR02_BNAME']
    ZV_ST_FROM = ZV_DI_ROW['UST12_VON']
    ZV_ST_TO   = ZV_DI_ROW['UST12_BIS']
    ...
```

In the above code, we use the named=True. named=True causes the row to be returned as a dictionary, so that the items in each column can be accessed based on the column name. If we do not use named = true, then iter_rows() would return a tuple, rather than a dictionary. In this case, we would access the values based on the position in the tuple, which would reflect the position of the column in the original dataframe:

```
for ZV_TU_ROW in ZV_DF.iter_rows():
    ZV_ST_USER = ZV_TU_ROW(0)
    ZV_ST_FROM = ZV_TU_ROW(1)
    ZV_ST_TO   = ZV_TU_ROW(2)
    ...
```

A Tuple is the same as a List, with the exception that a Tuple is not mutable. If an object is not mutable, it means that it cannot be changed after creation.

Dataframe: concatenate

Concatenate a list of dataframes

We often want to create multiple dataframes for different scenarios, and then concatenate the results into one dataframe.

For example, we might want to get the start date that a purchase order was blocked.

However, there are many diferent block flags for purchase orders. Therefore, we might want to do the same logic of getting the start date for each of these block flags.

Instead of writing out the same code for each block flag, we can loop around a list of block flags and store the block flag name and start date in a dataframe on each iteration of the loop. As the loop iterates, we can store these dataframes in a list of dataframes.

Once our loop finished, we can then append all of our dataframes that we find in our list, into one final dataframe.

```
ZV_LI_DFS = []
for ZV_ST_CDPOS_TABNAME_FNAME in ZV_LI_BLOCK_FLAGS:
    ZV_ST_FLAG_START = f'{ZV_ST_CDPOS_TABNAME_FNAME}_START'
    ZV_DF_CDPOSCDHDR_START = (
        ZV_DF_CDPOSCDHDR
        .filter(
            (
                PI_POLARS.col('ZF_CDPOS_TABNAME_FNAME') ==
                ZV_ST_CDPOS_TABNAME_FNAME
            ) &
            (PI_POLARS.col('CDPOS_VALUE_OLD') == '') &
            (PI_POLARS.col('CDPOS_VALUE_NEW') == 'X')
        )
        .with_columns(
            PI_POLARS.col('CDHDR_UDATE')
            .alias(f'{ZV_ST_FLAG_START}')
        )
    )
    ZV_LI_DFS.append(ZV_DF_CDPOSCDHDR_START)

ZV_DF_CDPOSCDHDR_START = (
    PI_POLARS.concat(
        ZV_LI_DFS,
        how = 'diagonal_relaxed'
    )
)
```

In the above code, we use the Polars function concat() to concatenate the list of dataframes into one dataframe. We use the keyword argument (kwarg) `'diagonal_relaxed'`, which means that extra columns are added if the dataframes do not have the same columns, with NULL values added as placeholders, for the rows coming from dataframes that did not have the extra columns.

Column: implode

List out information

If we want to turn a column into a list, we can use implode(). This is similar to .str.join(´_´), except for the following:

- With implode() the items remain separate and additional logic can be applied to them
- implode() is not limited to items that are of datatype string

```
ZV_DF_CDPOSCDHDR_IBAN_MULTICHG = (
    ZV_DF_CDPOSCDHDR_IBAN
    .sort(
       [
         'LFA1_LIFNR', # Supplier number
         'CDHDR_UDATE' # Change date
       ]
     )
    .group_by('LFA1_LIFNR')
    .agg(
      [
        PI_POLARS.col('CDDHDR_UDATE')
        .implode()
        .alias('ZF_LI_CDHDR_UDATE'),

        PI_POLARS.col('CDHDR_OBJECTID') # IBAN
        .implode()
        .alias('ZF_LI_CDHDR_OBJECTID'),

        PI_POLARS.col('CDHDR_OBJECTID')
        .n_unique()
        .alias('ZF_NU_CDHDR_OBJECTID_COUNTU')
      ]
    )
    .filter(
        PI_POLARS.col('ZF_NU_CDHDR_OBJECTID_COUNTU') > 1
    )
)
```

If we export `ZV_DF_CDPOSCDHDR_IBAN_MULTICHG` to Excel, Excel might replace some of our list columns with null. This is a current known Polars bug. To avoid this we can convert the lists to strings:

.list.join(' | ') only works if the items in the list are of type string. If the items are of type date or numeric, then we can use .list.eval() to cast them to string: Note: PI_POLARS.element() is used to obtain each element of a list.

```
ZV_DF_CDPOSCDHDR_IBAN_MULTICHG = (
   ZV_DF_CDPOSCDHDR_IBAN_MULTICHG
   .with_columns(
    [
        PI_POLARS.col('ZF_LI_DT_CDHDR_UDATE')
        .list.eval(PI_POLARS.element().dt.strftime('%Y%m%d'))
        .list.join('|')
        .alias('ZF_ST_CDHDR_UDATE_LIST')
    ]
)
```

Dataframe: unpivot

Convert a pivot table to a vertical table (unpivot)

The totals tables in the ECC version of SAP: trial balance (GLT0), supplier totals (LFC1), customer totals (KNC1), fixed assets totals (ANC1) are in pivot table format: values per month as separate month columns.

We often want to convert them from a pivot table to a vertical table: one month column and one value column.

Having these tables in a vertical format, makes it easier for us to make a line chart showing the balance / values per month:

```
ZV_LI_ST_COL_NAMES = ['GLT0_HSL01', ..., 'GLT0_HSL16']

ZV_DF_MELTED = (
    ZV_DF
    .melt(
        id_vars = [
            'GLT0_BUKRS',
            'GLT0_RYEAR',
            'GLT0_RACCT',
            'GLT0_HSLVT'
        ],
        value_vars = ZV_LI_ST_COL_NAMES,
        variable_name = 'ZF_GLT0_PERIOD',
        value_name = 'ZF_GLT0_HSLXX'
    )
)
```

In the above code we can use the function melt() to convert from a pivot table to a vertical table.

- id_vars: is the list of columns that will not be affected
- value_vars: is the list of columns that are to be converted into two new columns:
    - variable_name: is the name for the new column that will hold the name of the original column
    - value_name: is the name for the new column that will hold the value that was in the original column

Columns: select/ rename

Select fields

We can limit the number of fields in our dataframe to make it lighter:

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .select(
        [
            'EKPO_EBELN',
            'EKPO_EBELP',
            'EKPO_MATNR',
            'EKPO_NETWR',
            'EKPO_NETPR'
        ]
    )
)
```

Rename fields

Sometimes we need to rename fields:

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .rename(
        {
            'EBELN':'EKPO_EBELN',
            'EBELP':'EKPO_EBELP',
            'MATNR':'EKPO_MATNR',
            'NETWR':'EKPO_NETWR',
            'NETPR':'EKPO_NETPR'
        }
    )
)
```

Columns: drop

Drop one or more columns

We can drop columns that we no longer need. This keeps the dataframe light and makes the output easier to review. We can also drop columns to avoid confusion when joining multiple tables-

Use a list of column names with .drop(). This is the opposite of .select(): instead of saying which columns to keep, we say which columns to remove.

```
ZV_DF = (
    ZV_DF
    .drop(
        [
            'BSEG_XREF1',
            'BSEG_XREF2',
            'BSEG_XREF3'
        ]
    )
)
```

Dataframe: add row number

Add a case number / row ID

We often want to give each row a unique number. For example, to label each exception with a case number in the results table that we present in the dashboard.

Use .with_row_index(). offset=1 means that the first row will get the number 1 (instead of 0).

```
ZV_DF = (
    ZV_DF
    .with_row_index(
        name = 'ZF_NU_CASE_NUMBER',
        offset = 1
    )
)
```

Row number inside with_columns()

If we need the row number alongside other new columns inside the same with_columns(), we can use PI_POLARS.arange() based on the height of the dataframe (height is the number of rows).

```
ZV_DF = (
    ZV_DF
    .with_columns(
        [
            PI_POLARS.arange(1, ZV_DF.height + 1)
            .cast(PI_POLARS.Utf8)
            .alias('ZF_ST_CASE_NUM')
        ]
    )
)
```

Columns: slice/ conditional

Keep a part of a column

We often want to create a column that contains only part of the original column:

```
ZV_DF_EKPO = (
    ZV_DF_EKPO
    .with_columns(
        PI_POLARS.col('EKKO_ERDAT')
        .str.slice(0, 4)
        .alias('ZF_EKKO_ERDAT_YEAR')
    )
)
```

Create conditional columns

We often want to create columns, based on criteria:

```
ZV_DF_BSEG = (
    ZV_DF_BSEG
    .with_columns(
        PI_POLARS.when(
            PI_POLARS.col('BSEG_SHKZG') == 'S'
        )
        .then(
            PI_POLARS.col('BSEG_DMBTR')
        )
        .when(
            PI_POLARS.col('BSEG_SHKZG') == 'H'
        )
        .then(
            PI_POLARS.col('BSEG_DMBTR') * -1
        )
        .otherwise(
            PI_POLARS.lit(0)
        )
        .alias('ZF_BSEG_DMBTR_SIGNED')
    )
)
```

Column: string contains / starts_with

Filter where column contains a substring

We often want to find rows where a column contains a particular word or pattern. For example, we may want to find any SAP authorization value that contains a wildcard '*'. We can use .str.contains(), ***which accepts a regex string pattern***.

Note that we use r'*' (a raw string) so that python does not interpret the backslash. The * tells the regex to look for a literal *, not the regex meta-character *.

```
ZV_DF = (
    ZV_DF
    .filter(
        PI_POLARS.col('UST12_VON')
        .str.contains(r'\*')
    )
)
```

In the above code, r'*' represents two layers:

- r'…' – **python layer**: r: raw string: raw string means ***do not*** interpret at python layer:
    - For example, python will consider '\n' as newline.. and r'\n' as the two characters \ and n.
    - If we do not say r'', then python can accidentally remove our \, before we get to the regex layer
- * - **regex layer**: \: escape: ***do not*** interpret at regex layer:
    - For example, regex will consider * as: anything containing *
    - If we do not put \, then regex will consider * as meaning “allow for repeated characters”:
        - For exemple: str.contains(r' (ab)\1*') will match 'ab', 'abab', 'abababab'

So to summarize, because str.contains() expects a regex string…. we use **r'…'** to mean, ***do not interpret at the python level*** and **** to mean, **do not interpret at the regex level**. This way we can say “does my string contain *?”

Since we are just looking for cases that contain *, we can also use find() or contains() with literal = True:

```
ZV_DF = (
    ZV_DF
    .filter(
        PI_POLARS.col('UST12_VON')
        .str.find('*')
    )
)
```

```
ZV_DF = (
    ZV_DF
    .filter(
        PI_POLARS.col('UST12_VON')
        .str.contains('*', literal=True)
    )
)
```

Filter where column starts with a character

starts_with() checks the beginning of the string. For example, SAP authorization placeholders often start with '$':

```
ZV_DF = (
    ZV_DF
    .filter(
        PI_POLARS.col('USOBT_LOW')
        .str.starts_with('$')
    )
)
```

Column: regex from a keyword list

Build a regex expression from a list of keywords

We often want to search for any of many keywords entered by the auditor. For example, suspicious words such as 'cash', 'gift' or 'bonus' in the journal entry text.

We use PI_RE.escape() so that special regex characters in the keywords are treated literally (for exemple, a '.' in a keyword stays a literal '.', not 'any character'). If we have a special character, then PI_RE.escape will replace it with \. For exemple, 'bonus?' would become 'bonus\? '. Then, when we pass the string to str.contains(), it will search for 'bonus?' and not interpret the '?' as a special regex character.

The escaped key words, shown in the ZV_LI_KEYWORDS variable below, are joined with '|', giving 'cash|gift|bonus'. str.contains() interprets this as a regex pattern, that means: “Return True if you find the words cash or gift or bonus in the text”.

In this way, our filter is matching any of our key words, including any characters that are not alpha-numeric, such as '?'.

```
import re as PI_RE

ZV_LI_KEYWORDS = ['cash', 'gift', 'bonus']

ZV_ST_REGEX = '|'.join(
    [PI_RE.escape(k) for k in ZV_LI_KEYWORDS]
)

ZV_DF = (
    ZV_DF
    .filter(
        PI_POLARS.col('BKPF_BKTXT')
        .str.contains(ZV_ST_REGEX)
    )
)
```

A fun site for learning regex: **https://regexone.com/**

Columns: string <-> numeric conversion

String from numeric

The most common type of string datatype that we use is Utf8 (which is basically ASCII). For example, if we wanted to do a Benford's law statistical analysis on the leading digits of our numbers in the data set, we might use the following to first convert our numeric field into a string:

```
ZV_DF_BSAK = (
    ZV_DF_BSAK
    .with_columns(
        PI_POLARS.col('BSAK_DMBTR')
        .cast(
            PI_POLARS.Utf8,
            strict=False
        )
        .str.slice(0, 2)
        .alias('ZF_BSAK_DMBTR_2_DIG')
    )
)
```

For cast and strptime (and many other functions in Polars), we use strict = False to get a null value, rather than having our program bug, in case the number cannot be interpreted.

Numeric from string

If we have a column that is in string format - for example, just after importing data from a text file - then we can convert it to numeric format, if it is listed in the fields to be converted to numeric format:

```
ZV_DF = (
    ZV_DF
    .with_columns(
        [
            PI_POLARS.col(ZV_ST_COL)
            .cast(
                PI_POLARS.Float64,
                strict=False
            )
            for ZV_ST_COL in ZV_LI_NUMERIC_COLS
        ]
    )
)
```

Columns: string <-> date conversion

String from date

If we have a column in date format, we may wish to convert it to string format, so that we can extract certain information, such as the year or the month, or use the field as a join key.

```
ZV_DF_BSID = (
    ZV_DF_BSID
    .with_columns(
        PI_POLARS.col('ZF_BSID_BUDAT_DT')
        .dt.strftime('%Y%m%d')
        .alias('ZF_BSID_BUDAT_ST')
    )
)
```

Date from string

As part of our naming convention, we decided to always import dates as strings in the format `'YYYYMMDD'`. However, if we want to calculate the difference in days between two dates, we need to convert to date.

```
ZV_DF_BSID = (
    ZV_DF_BSID
    .with_columns(
        PI_POLARS.col('BSID_BUDAT')
        .str.strptime(
            PI_POLARS.Date,
            '%Y%m%d',
            strict=False
        )
        .alias('ZF_BSID_BUDAT_DT')
    )
)
```

Always import dates as 'YYYYMMDD'

In our naming convention, we always import date fields as strings in the format 'YYYYMMDD'. This is so that we can be consistent when writing code concerning dates.

If we create date variables, we also always create them as strings in the format 'YYYYMMDD'.

Columns: calculate age

Calculate age

In the below code, we calculate, what Polars calls a Duration: a difference between two dates. We then use the method .dt.total_days() to convert this duration into days.

```
from datetime import date as PI_DATE

ZV_DT_TODAY = PI_DATE.today()
ZV_DF_EKKO = (
    ZV_DF_EKKO
    .with_columns(
        PI_POLARS.col('EKKO_BEDAT')
        .str.strptime(
            PI_POLARS.Date,
            '%Y%m%d',
            strict=False
        )
        .alias('ZF_EKKO_BEDAT_DT')
    )
    .with_columns(
        (
            PI_DATE.today() -
            PI_POLARS.col('ZF_EKKO_BEDAT_DT')
        )
        .dt.total_days()
        .alias('ZF_EKKO_BEDAT_DT_AGE')
    )
)
```

In the above code, we convert the Duration into days. However, we could have also converted it into hours, minutes or seconds instead:

```
        (
            PI_DATE.today() -
            PI_POLARS.col('ZF_EKKO_BEDAT_DT')
        )
        .dt.total_hours()
        .alias('ZF_EKKO_BEDAT_AGE_HOURS')
```

```
        (
            PI_DATE.today() -
            PI_POLARS.col('ZF_EKKO_BEDAT_DT')
        )
        .dt.total_minutes()
        .alias('ZF_EKKO_BEDAT_AGE_MINS')
```

```
        (
            PI_DATE.today() -
            PI_POLARS.col('ZF_EKKO_BEDAT_DT')
        )
        .dt.total_seconds()
        .alias('ZF_EKKO_BEDAT_AGE_SECS')
```

Columns: compare rows

Current row minus previous row

We often need to compare to a prior row. We can do this using .diff(). For example, if we want to compare the date of our purchase order to the date on the previous row, after sorting.

```
ZV_DF_EKPO_EKKO = (
    ZV_DF_EKPO_EKKO
    .sort(
        [
            'EKPO_WERKS',
            'EKKO_LIFNR',
            'EKPO_MATNR',
            'ZF_EKKO_BEDAT_DT'
        ]
    )
    .with_columns(
        PI_POLARS.col('ZF_EKKO_BEDAT_DT')
        .diff()
        .over(
            [
                'EKPO_WERKS',
                'EKKO_LIFNR',
                'EKPO_MATNR'
            ]
        )
        .dt.total_days()
        .fill_null(0)
        .alias('ZF_EKKO_BEDAT_DAYS_FROM_PREV')
    )
)
```

In the above code: we use .over([]) to make sure that we do not compare dates between different plants, suppliers, materials.

For the first row for each plant, supplier, material we would get null(). We may not want null, because null can disrupt other analysis. Therefore, we can say .fill_null(0), to replace the null with 0.

Columns: compare rows

Reference another row

We often need to reference a previous row. For example, we may wish to know if we have a supplier that appears to be a shell company, because they are sending invoices that follow each other incrementally.

```
ZV_DF_BSAIK = (
    ZV_DF_BSAIK
    .with_columns(
        PI_POLARS.col('BSAIK_XBLNR')
        .cast(PI_POLARS.Utf8)
        .str.strip_chars()
        .str.extract_all(r'(\d+)')
        .list.last()
        .cast(PI_POLARS.Int64, strict=False)
        .alias('ZF_BSAIK_XBLNR_NUM')
    )
    .sort(
        [
            'BSAIK_BUKRS',
            'BSAIK_LIFNR',
            'BSAIK_BLDAT'
        ]
    )
    .with_columns(
        (
            PI_POLARS.col('ZF_BSAIK_XBLNR_NUM') -
            (
                PI_POLARS.col('ZF_BSAIK_XBLNR_NUM')
                .shift(1)
                .over(
                    [
                        'BSAIK_BUKRS',
                        'BSAIK_LIFNR'
                    ]
                )
            )
        )
        .alias('ZF_BSAIK_XBLNR_NUM_MINUS_PREV')
    )
)
```

In the above code, str.extract_all() takes a raw string and interprets it as a regex to provide a list of all the strings that matched the pattern. The regex pattern \d* means “Take any strings that are one or more digits”. Then we apply to the list: .list.last(), which means: “Only take the last item of the list”.

Here we are assuming that the supplier might have varying invoice patterns, but that it is the last numeric part of the invoice number that is incremented, each time they issue an invoice.

We may wish to try other assumptions, such as “the first numeric part of the invoice number that is incremented”.

Columns: expression

Create an expression from a dictionary

The below code creates an expression based on information in a dictionary:

```
ZV_OB_EXPR_BUCKET = None

for ZV_DI_LIMIT in ZV_LI_DI_APPROVAL_LIMITS:
    ZV_ST_LEVEL = ZV_DI_LIMIT['level']
    ZV_NU_THRESHOLD = ZV_DI_LIMIT['threshold_USD']
    ZV_ST_OPERATOR  = ZV_DI_LIMIT['operator']

    if ZV_ST_OPERATOR == '<=':
        ZV_BO_CONDITION = (
            PI_POLARS.col('EKPO_NETWR') <= ZV_NU_THRESHOLD
        )

    elif ZV_ST_OPERATOR == '>':
        ZV_BO_CONDITION = (
            PI_POLARS.col('EKPO_NETWR') > ZV_NU_THRESHOLD
        )

    else:
        continue

    if ZV_OB_EXPR_BUCKET is None:
        ZV_OB_EXPR_BUCKET = (
            PI_POLARS.when(ZV_BO_CONDITION)
            .then(
                PI_POLARS.lit(ZV_ST_LEVEL)
            )
        )

    else:
        ZV_OB_EXPR_BUCKET = (
            ZV_OB_EXPR_BUCKET
            .when(ZV_BO_CONDITION)
            .then(
                 PI_POLARS.lit(ZV_ST_LEVEL)
            )

ZV_OB_EXPR_BUCKET = (
       ZV_OB_EXPR_BUCKET
       .otherwise(
           PI_POLARS.lit('OutOfRange')
       )
```

Create a column using an expression

The below code creates a column based on the expression above:

```
ZV_DF_EKPOEKKO = (
    ZV_DF_EKPOEKKO
    .with_columns(
        [
            ZV_OB_EXPR_BUCKET
            .alias('ZF_ST_APPROVAL_BUCKET')
        ]
    )
)
```

Columns: concatenate

Concatenate fields together

We often want to concatenate string fields. For example, to provide the description for a code: Assuming we have added the field TSTCT_TTEXT_BKPF to the BSEG/BKPF cube, we can concatenate the transaction code with its description:

```
B01_12_IT_DF_BSEGBKPF_CUBE = (
    B01_12_IT_DF_BSEGBKPF_CUBE
    .with_columns(
        PI_POLARS.concat_str(
            [
                PI_POLARS.col('BKPF_TCODE'),
                PI_POLARS.col('TSTCT_TTEXT_BKPF')
            ],
            separator='_'
        )
        .alias('ZF_BKPFTCODE_TSTCTTTEXT')
    )
)
```

Columns: cum_sum(): window

Create a window with cum_sum()

We may wish to know if two purchase orders are within the same window. For example, they are close in days to each other.

If there are purchase orders that are close in date range and that have similar information, then it could be that the purchase orders have been split in order to circumvent delegation of authority controls.

We can use cum_sum() to add up the values from previous rows. Since True is also equal to 1, cum_sum() below increments for each new window.

```
ZV_DF_EKPO_EKKO = (
    ZV_DF_EKPO_EKKO
    .sort(
        [
            'ZF_LIFNRERNAMMATNRWERKSAPPBUCK',
            'ZF_EKKO_BEDAT_DT'
        ]
    )
    .with_columns(
        (
            (
                (
                    (
                        PI_POLARS.col('ZF_EKKO_BEDAT_DT').diff()
                        .dt.total_days()
                    )
                    .fill_null(0)
                    .abs()
                ) >= int(ZV_NU_WINDOWDAYS)
            ) |
            (
                PI_POLARS.col('ZF_LIFNRERNAMMATNRWERKSAPPBUCK') !=
                (
                    PI_POLARS.col('ZF_LIFNRERNAMMATNRWERKSAPPBUCK')
                    .shift(1)
                )
            )
        )
        .cum_sum()
        .alias('ZF_NU_CASE_NUMBER')
    )
)
```

In the above code, .diff() gives us the Duration between the date on the current row compared to the previous row. .dt.total_days() converts that Duration into the number of days. In case it was the first row in the dataframe, .fill_null(0) replaces the null value with 0. If the number of days exceeds the ZV_NU_WINDOWDAYS threshold or the supplier/user/material/plant or approval bucket is diferente, then the result is True… which is also equal to 1. .cum_sum() therefore, adds 1 to the field ZV_NU_CAS_NUMBER.

Meaning the case number is incremented each time there is a difference in days with the previous row that is greater than the threshold or when it is a different supplier (LIFNR)/ user(ERNAM)/ material (MATNR)/ plant (WERKS)/ approval bucket.

Columns: cum_sum(): running total

Reverse running total with cum_sum(), over

We often want to know what the running total is in our dataset. For example, if we want to know the total movements to-date.

In the below example, we use a reverse running total to know the total future stock-out movements. This can be useful to see how much of the stock is used in the future.

For example, we might want to see how much of the stock purchased is used in the future.

```
ZV_DF_MSEG_MKPF_TOT_MATNR_MTH = (
    ZV_DF_MSEG_MKPF_TOT_MATNR_MTH
    .sort(
        [
            'MSEG_BUKRS',
            'MSEG_MATNR',
            'ZF_MKPF_BUDAT_YRMTH'
        ]
    )
    .with_columns(
        [
            PI_POLARS.col('ZF_MSEG_DMBTR_CREDIT')
            .cum_sum(reverse=True)
            .over(
                [
                    'MSEG_BUKRS',
                    'MSEG_MATNR'
                ]
            )
            .alias('ZF_MSEG_DMBTR_CREDIT_FUTURE')
        ]
    )
)
```

In the above code, to get the reverse running total, we use cum_sum() with reverse = True.

We also use over([]) to only accumulate within the same company (MSEG_BUKRS) and material (MSEG_MATNR).

Column: explode

Create a list of words, then rows from a string field

We often want to get one word from a sentence, phrase, or name. For example, in SAP ECC we have the supplier name in LFA1_NAME1. We often want to compare this name to the names in the list of personnel. We can use explode() to get one row per word in the NAME1 column. We then use explode() to get one row per word in the name from personnel. Then we can use join() with how='inner' to filter on matched words.

```
ZV_DF_LFA1 = (
    ZV_DF_LFA1
    .with_columns(
        PI_POLARS.col('LFA1_NAME1')
        .str.split(' ')
        .alias('ZF_LI_LFA1_NAME1')
    )
    .explode('ZF_LI_LFA1_NAME1')
    .rename(
        {'ZF_LI_LFA1_NAME1' : 'ZF_ST_LFA1_NAME1_EXPLODE'}
    )
)
```

The above code creates a column that contains all of the words in LFA1_NAME1 in the form of a list. A new column is created using the .alias() method.

The new list column is then exploded: for each item, a new row is created. The column is then renamed, so that it reflects our naming convention.

Column/ variable: zone an string/ number

Column: zone a text column

For columns, we can use the Polars method .str.zfill(8) to zone the column.

```
ZV_DF = (
    ZV_DF
    .with_columns(
        PI_POLARS.col('BSEG_HKONT')
        .str.zfill(8)
    )
)
```

Column: zone a numeric column

If our column is a numeric type, we need to first convert it to string type:

```
ZV_DF = (
    ZV_DF
    .with_columns(
        PI_POLARS.col('BSEG_DMBTR')
        .cast(PI_POLARS.Utf8)
        .str.zfill(8)
    )
)
```

Variable: zone a text item

If our list of accounts are strings, then we can use the built-in python method for strings: zfill(), to format with leading zeros. For example, zfill(8): format with leading zeros and length of 8 characters.

```
ZV_LI_ST_CASH_ACCOUNTS = ['56789', '78923']

ZV_LI_ST_CASH_ACCOUNTS = [
    ZV_ST_ACCOUNT.zfill(8)
    for ZV_ST_ACCOUNT in ZV_LI_ST_CASH_ACCOUNTS
]
```

Variable: zone an integer item

We often want to pad an integer with zeros. For example, if you have a list of general ledger account numbers –for cash, then you might want to use these to filter your general ledger. This is because the general ledger account field in SAP has leading zeros.

If your list of cash accounts, is a list of integers, you would convert it to string and add leading zeros using the built-in string formatting syntax used in f-strings. For example, f'{…:08}': format with leading zeros and length of 8:

```
ZV_LI_NU_CASH_ACCOUNTS = [56789, 78923]

ZV_LI_ST_CASH_ACCOUNTS = [
    f'{ZV_NU_ACCOUNT:08}'
    for ZV_NU_ACCOUNT in ZV_LI_NU_CASH_ACCOUNTS
]
```

Variables - conversions

When converting variables, we can use the built-in python or plain python libraries. No python library installs required.

Variable: string from numeric

We may want to convert a numeric variable to a string.

```
ZV_NU_AMOUNT = 12345.67
ZV_ST_AMOUNT = str(ZV_NU_AMOUNT)
```

Variable: integer from string

We may want to convert a string variable to an integer.

```
ZV_ST_THRESHOLD = '500000'
ZV_NU_THRESHOLD = int(ZV_ST_THRESHOLD)
```

Variable: float from string

We may want to convert a string variable to a float.

```
ZV_ST_THRESHOLD = '500000.67'
ZV_NU_THRESHOLD = float(ZV_ST_THRESHOLD)
```

Variable: string from date

We can use the built-in python .strftime() function to convert a python date variable to a string. Note that %Y means YYYY, as opposed to YY.

```
ZV_ST_END_DATE = ZV_DT_END_DATE.strftime('%Y%m%d')
```

Variable: date from string

We can use the python datetime library and function datetime, which has the function strptime() to convert a string variable to a date. Note that %Y means YYYY, as opposed to YY.

```
from datetime import datetime as PI_DATETIME

ZV_DT_DATE = (
    PI_DATETIME.strptime(
        ZV_ST_DATE,
        '%Y%m%d'
    )
    .date()
)
```

List – access items/ loop

Access an object in a list

The below code gives us the second item in the list:

```
ZV_OB_MY_OBJECT = ZV_LI_MY_LIST[1]
```

Access all objects in a list one-by-one

We can use a for loop to access objects in a list one-by-one. The below code accesses each dictionary of approval limits in our list of approval limits. The code then accesses each value inside each dictionary:

```
for ZV_DI_LIMIT in ZV_LI_DI_APPROVAL_LIMITS:
    ZV_ST_LEVEL     = ZV_DI_LIMIT['level']
    ZV_NU_THRESHOLD = ZV_DI_LIMIT['threshold_USD']
    ZV_ST_OPERATOR  = ZV_DI_LIMIT['operator']
    print(
        f'Level: {ZV_ST_LEVEL}, '
        f'Threshold: {ZV_NU_THRESHOLD}, '
        f'Operator: {ZV_ST_OPERATOR}'
    )
```

List – zip - access items in parallel lists

Zip: Simultaneously access information from multiple lists, same index

Sometimes we have more than one list, and we want to access the information from the same position in each list at the same time, so that we can compare the values, or so that we can group logical steps together, rather than repeating code. We can use zip, to put the values from the different lists, side-by-side according to their index (position).

For example, the below code takes two lists of field names: one for purchase order block start date and one for purchase order block end date. The lists are in the same order of block flag type (for example, purchasing block, posting block, etc.).

The zip() puts the lists side-by-side, so that we can access the field names as pairs from within the for loop. We then use the column names as variables to bring up the columns for block start date and block end date… one block flag at a time.

This way, we can compare the block start and block end date to the purchase order date. We use this approach to see if there are any purchase orders between block start and end dates for the same block flag.

```
ZV_LI_DF = []
for ZV_ST_ST, ZV_ST_END in zip(ZV_LI_COL_ST, ZV_LI_COL_END):
    ZV_DF = (
        ZV_DF_EKPO_EKKO
        .filter(
            (
                PI_POLARS.col('EKKO_BEDAT') >=
                PI_POLARS.col(ZV_ST_ST)
            ) &
            (
                PI_POLARS.col('EKKO_BEDAT') <=
                PI_POLARS.col(ZV_ST_END)
            ) &
            (
                PI_POLARS.col(ZV_ST_ST).is_not_null()
            ) &
            (
                PI_POLARS.col(ZV_ST_END).is_not_null()
            )
        )
    )
    ZV_LI_DF.append(ZV_DF)

ZV_DF_EKPO_EKKO_WHEN_BL = (
    PI_POLARS.concat(
        ZV_LI_DF,
        how='vertical'
    )
)
```

The above code creates a dataframe on each iteration of the for loop. Each dataframe is a subset of the original, based on the filter applied. Each dataframe created is stored in a list of dataframes. After the for loop has finished, these dataframes are then all appended together using the concat function. In the concat function we use the keyword argument (kwarg) vertical. Vertical means that python expects the list of fields for each dataframe created to be the same, which is True here because they are all created from the same source dataframe.

Dictionaries – loop

Access all objects in a dictionary one-by-one

We can use:
for ZV_ST_KEY, ZV_DI in <dictionaryName>.items():
to loop through all dictionaries inside a dictionary.

```
for ZV_ST_KEY, ZV_DI_DEPARTMENT in ZV_DI_COMPANY.items():
    for ZV_ST_KEY, ZV_DI_EMPLOYEE in ZV_DI_DEPARTMENT.items():
        ZV_ST_NAME   = ZV_DI_EMPLOYEE['name']
        ZV_ST_ROLE   = ZV_DI_EMPLOYEE['role']
        ZV_NU_SALARY = ZV_DI_EMPLOYEE['salary']
```

The above code accesses each department dictionary one-by-one.

Within each department dictionary, the code then accesses each employee dictionary.

Within each employee dictionary, the code then accesses the values identified by the keys: name, role and salary.

If we are not sure if a key exists, we can use ZV_DI_EMPLOYEE.get('name'). If the key does not exist, this will return None, instead of crashing the program.

Access an object in a dictionary

The below code creates a dictionary that has three levels.

```
ZV_DI_MY_DICTIONARY = {
    'key': {
        'key': {
            'key': myObject
        }
    }
}
```

The below code gives us the object "myObject" that we mention above:

```
ZV_OB_MY_OBJECT = ZV_DI_MY_DICTIONARY['key']['key']['key']
```

Dictionaries to dataframes

Convert dictionaries of lists to a dataframe

We can convert a dictionary to a dataframe. For example, if the dictionary is in the format:

```
ZV_DI_LI_CUSTOMERS = {
    'Name':    ['Claire', 'John'],
    'Country': ['Portugal', 'UK'],
}
```

The below code will convert this dictionary to a dataframe with two columns: Name | Country.

```
ZV_DF = PI_POLARS.DataFrame(ZV_DI_LI_CUSTOMERS)
```

Convert lists of nested dictionaries to a dataframe -> structures

If we have a list of nested dictionaries:

```
ZV_LI_DI_CUSTOMERS = [
    {
        'Name': 'Claire',
        'Address': {
            'City':    'Lisbon',
            'Country': 'Portugal'
        }
    },
    {
        'Name': 'John',
        'Address': {
            'City':    'London',
            'Country': 'UK'
        }
    }
]
ZV_DF = PI_POLARS.DataFrame(ZV_LI_DI_CUSTOMERS)
```

After the above code, we will have a dataframe, with the columns Name | Address. However, the Address column will be a structure. A structure is like a dictionary inside a column item. We can convert the structure into two columns: City | Country:

```
ZV_DF = ZV_DF.unnest('Address')
```

Structures are also sometimes used in dataframes, in order to hold two or more pieces of information together for further processing. See section on Levenshtein where we use structures.

Download URL

Download information from a website

We often want to download information from a website, such as sanctions lists:

```
import requests as PI_REQUESTS
import io as PI_IO

def FC_DOWNLOAD_CSV_FROM_URL(ZVFCI_FILENAME):
    ZV_URL = (
        's://sanctionslistservice.ofac.treas.gov'
        f'/api/download/{ZVFCI_FILENAME}'
    )
    ZV_RESPONSE = PI_REQUESTS.get(ZV_URL)
    ZV_RESPONSE.raise_for_status()
    return PI_IO.StringIO(
        ZV_RESPONSE.content.decode('iso-8859-1')
    )

ZV_DF = PI_Pandas.read_csv(
    FC_DOWNLOAD_CSV_FROM_URL('SDN.csv'),
    delimiter = ',',
    header = None,
    names = [
        'SDN_ENT_NUM',
        'SDN_NAME',
        ...
    ]
)
ZV_DF = PI_POLARS.from_Pandas(ZV_DF)
```

The above code reads the website CSV into a Pandas dataframe. The Pandas dataframe is then converted into a Polars dataframe for further processing. We convert the Pandas dataframe into a Polars dataframe, because Polars can handle large data sets quicker.

Download FX rates

Get daily FX rates from Yahoo Finance

We often need historical FX rates to revalue foreign-currency postings. The yfinance library lets us download daily closing prices for any ticker between two dates.

yfinance returns a Pandas dataframe. We can then loop the rows and build a dictionary that has the date (as a string in YYYYMMDD format) as the key and the closing rate as the value.

progress=False stops yfinance from printing a download progress bar to the terminal.

```
import yfinance as PI_YFINANCE

ZV_DF_DATA = PI_YFINANCE.download(
    'EURUSD=X',
    start = ZV_DT_BEG_DATE,
    end   = ZV_DT_END_DATE,
    interval = '1d',
    progress = False
)

ZV_DI_EXC_RATE = {
    idx.strftime('%Y%m%d'): row['Close']
    for idx, row in ZV_DF_DATA.iterrows()
}
```

In the above code we can see that we use idx.strftime. The result from yFinance is a Pandas dataframes. Pandas dataframe rows all have an index name. In the case of the Pandas dataframe from yFinance, the idx name is the date column. The column Close is the column that contains the rate at the end of the day.

Regex pattern matching

Keep only alpha numeric

We often want to tidy up names before we do comparisons. For this we can use regex:

```
ZV_DF = (
    ZV_DF
    .with_columns(
        PI_POLARS.col('SDN_NAME')
        .str.strip_chars()
        .str.replace_all(
            r'[^0-9a-zA-Z]',
            ''
        )
        .str.to_uppercase()
        .alias('ZF_SDN_NAME_CLEAN')
    )
)
```

Levenshtein / map elements

Compare text from two columns: levenshtein

We often want to compare data from two columns. For example, we might want to compare two addresses. However, levenshtein.distance() can only work on variables, not on columns. Therefore, we put the values on each row in a structure (which is kind of like a mini dictionary in each item of a column) and pass each structure to the function using .map_elements():

```
import Levenshtein as PI_LEVENSHTEIN

def FC_INV_LEVENSHTEIN_DISTANCE(
    ZVFCI_ST_1,
    ZVFCI_ST_2
):
    if ZVFCI_ST_1 is None or ZVFCI_ST_2 is None:
        return None
    return round(
        (1 / (PI_LEVENSHTEIN.distance(
            ZVFCI_ST_1,
            ZVFCI_ST_2
        ) + 1)) * 100,
        2
    )

def FC_MAP_INV_LEVENSHTEIN_DISTANCE(ZVFCI_DI):
    return FC_INV_LEVENSHTEIN_DISTANCE(
        ZVFCI_DI['LFA1_NAME1'],
        ZVFCI_DI['SDN_NAME']
    )

ZV_DF_LFA1_OFACADD = (
    ZV_DF_LFA1ADD
    .with_columns(
        PI_POLARS.struct(
            [
                'LFA1_NAME1',
                'SDN_NAME'
            ]
        )
        .map_elements(
            FC_MAP_INV_LEVENSHTEIN_DISTANCE,
            return_dtype=PI_POLARS.Float64
        )
        .alias('ZF_INV_LEVENSHTEINDIST')
    )
)
```

Typical shared functions

Typical shared functions

The following code examples give tips for project infrastructure.

These code snippets are typically stored in shared functions that all scripts can access.

They are typically shipped with all projects, to allow for import, export, logging, monitoring, use of external variables, etc.

Start-time, End-time

Record the start and end time of your script

Scripts can take a long time to run, so we may wish to record the start and end times:

```
from datetime import datetime as PI_DATETIME

ZV_ST_STARTTIME = PI_DATETIME.now().strftime(
    '%Y-%m-%d %H:%M:%S'
)

# ...

ZV_ST_ENDTIME = PI_DATETIME.now().strftime(
    '%Y-%m-%d %H:%M:%S'
)

print(
    f'Script started at {ZV_ST_STARTTIME} and ended at {ZV_ST_ENDTIME}'
)
```

Import a text file

Create a file path, that works for Windows or Linux:

Given a variable for folder and file name, create a full path that is correct for both Windows (\) and Linux (/):

When we put \ in a string, python thinks it is an escape. For example \n: new line, \t: tab, \\ backslash. We can use r'' to make our string raw, so that the \ is not interpreted by python as escape. Note that we must not put the last \ when using raw for file-paths.

```
ZV_ST_RAW_PATH  = r'C:\folder'

ZV_ST_FILE_PATH = PI_OS.path.join(
    ZVFCI_ST_FOLDER,
    ZVFCI_ST_SOURCE_FILE
)
```

in the above example, we create the file path with the plain python os library´s path.join() function, in order to avoid differences when switching from Windows to Linux environments.

Import a text file:

When importing text files in python, we want to make sure that we do not get bugs due to special characters, or different encodings. We can use the below code example to avoid typical import errors.

```
def FC_IMPORT_TEXT(
    ZVFCI_ST_FILE_PATH,
    ZVFCI_ST_DELIMITER,
    ZVFCI_LI_ENCODINGS  # ['utf8', 'utf8-lossy', 'windows-1252']
):
    ZV_DF = None
    ZV_ER_LAST = None
    for ZV_ST_ENCODING in ZVFCI_LI_ENCODINGS:
        try:
            ZV_DF = PI_POLARS.read_csv(
                ZVFCI_ST_FILE_PATH,
                separator = ZVFCI_ST_DELIMITER,
                encoding = ZV_ST_ENCODING,
                ignore_errors = True,
                has_header = True,
                infer_schema_length = 0,
                null_values = [''],
                quote_char = None,
                truncate_ragged_lines = True,
                missing_utf8_is_empty_string = True
            ).fill_null('')
            break
        except Exception as ZV_ER:
            ZV_ER_LAST = ZV_ER
    if ZV_DF is None:
        print(f'Last error: {ZV_ER_LAST}')
    return ZV_DF
```

Import a text file – force column types

Obtain the list of columns in a dataframe

When python is importing, it can guess the column types. However, these are often, not what we require. Therefore, we can force the column types.

First, we get the list of columns:

```
ZV_LI_COLS = ZV_DF.columns
```

Create a dictionary with the type for each column

Next, we define the type for each column. For example, we can decide that all columns will have the type Utf8.

The below code creates a dictionary that has the key as the field name and the value 'Utf8'.

```
ZV_DI_COLS = {
    ZV_ST_COL_NAME: PI_POLARS.Utf8
    for ZV_ST_COL_NAME in ZV_LI_COLS
}
```

Use a dictionary to force column types

The ZV_DI_COLS can then be used to specify the type for those columns:

```
ZV_DF = PI_POLARS.read_csv(
    ZV_ST_FILE_PATH,
    separator = ZVFCI_ST_DELIMITER,
    encoding = ZV_ST_ENCODING,
    ignore_errors = True,
    has_header = True,
    schema_overrides = ZV_DI_COLS,
    null_values = [''],
    quote_char = None,
    truncate_ragged_lines = True,
    missing_utf8_is_empty_string = True
).fill_null('')
```

Note: If we want all columns to be Utf8, we can also use the keyword argument (kwarg) infer_schema_length=0.

Import a json

Import .json to dictionary

The below example uses the built in open() python function to create a file object. The code then uses the json library´s load function to load the file object as a dictionary.

```
import json as PI_JSON

ZV_ST_FILEPATH_APPROVAL_LIMITS = (
    PI_OS.path.join(
        ZF_ST_SOURCES_FOLDER,
        'A_APPROVAL_LIMITS.json'
    )
)

with open(
    ZV_ST_FILEPATH_APPROVAL_LIMITS,
    'r',
    encoding='utf-8'
) as ZV_OB_FILE:
    ZV_DI_APPROVAL_LIMITS = PI_JSON.load(ZV_OB_FILE)
```

In the above code, the built-in **with** keyword is used to enable a file to be opened and then closed once the operations on that file are completed for the open. The keyword **with** can be used in a similar way to allow a connection to a database to be opened... and then close the connection following the completion of the operations that are listed in the tab indent under the open():

.json file structure

A .json file is a text file that:

- has the extension .json, instead of .txt.
- starts and ends with the brackets {}.
- has content written as <keyName>: <object>
- <object> can be of any type:
  - <keyName>: "<text>"
  - <keyName>: [<listItem1, listItem2, listItem3>]
  - <keyName>: <number>
  - <keyName>: {<dictionary>}

In a typical JSON, a keyName like "Approval_limits" might point to an object of type list. Each object in the list is then of type dictionary.

Export to Excel

Export to Excel

We do most of our analysis in Polars. However, to export to Excel, we use a Pandas function.

The Pandas function will try to import the openpyxl library. Therefore, to run the below code, we need to ensure we have openpyxl in our virtual environment. We do not need to import openpyxl to our script because Pandas will import it automatically. As per our naming convention set-up rules, we should openpyxl in our requirements.txt.

```
pip install openpyxl
```

```
def FC_EXPORT_EXCEL(
    ZVFCI_DF_INPUT,
    ZVFCI_ST_RESULTS_FOLDER,
    ZVFCI_ST_RESULTS_FILE
):
    PI_OS.makedirs(
        ZVFCI_ST_RESULTS_FOLDER,
        exist_ok=True
    )
    ZV_ST_FILE_PATH = PI_OS.path.join(
        ZVFCI_ST_RESULTS_FOLDER,
        ZVFCI_ST_RESULTS_FILE
    )
    ZV_DF_Pandas = ZVFCI_DF_INPUT.to_Pandas()
    ZV_DF_Pandas.to_excel(
        ZV_ST_FILE_PATH,
        index = False,
        engine = 'openpyxl'
    )
```

Log memory usage

Log memory usage

Sometimes we want to analyze very heavy data, but our machine is too small and the RAM cannot handle the size. Therefore, we might want to log the RAM usage as the script progresses.

The below script takes the script name and step and outputs to the log the RAM used.

```
import psutil as PI_PSUTIL
import os as PI_OS
from Z_SHARED_FUNCTIONS.FC_LOGGER import ZV_OB_LOGGER

def FC_LOG_MEM_USAGE(ZVFCI_ST_SCRIPT, ZVFCI_ST_SCRIPT_STEP):
    ZV_PROCESS  = PI_PSUTIL.Process(PI_OS.getpid())
    ZV_MEM_PROC = ZV_PROCESS.memory_info().rss / (1024 ** 3)
    ZV_MEM_SYS  = PI_PSUTIL.virtual_memory()
    ZV_OB_LOGGER.info(
        f'Running: script: {ZVFCI_ST_SCRIPT}, {ZVFCI_ST_SCRIPT_STEP}'
    )
    ZV_OB_LOGGER.info(
        f'[PROCESS] Python process memory: {ZV_MEM_PROC:.2f} GB'
    )
    ZV_OB_LOGGER.info(
        f'[SYSTEM]  Total RAM:     {ZV_MEM_SYS.total     / (1024**3):.2f} GB'
    )
    ZV_OB_LOGGER.info(
        f'[SYSTEM]  Used RAM:      {ZV_MEM_SYS.used      / (1024**3):.2f} GB'
    )
    ZV_OB_LOGGER.info(
        f'[SYSTEM]  Available RAM: {ZV_MEM_SYS.available / (1024**3):.2f} GB'
    )
```

Free memory

Release RAM between heavy steps

After a big group_by() or join we often no longer need certain intermediate dataframes. Even though python will eventually free the memory, it does not always do this immediately. This can be a problem if our next step is also heavy, because the script may run out of RAM.

We can help python free the memory by:

- assigning the dataframes to None - this removes our reference to them
- calling PI_GC.collect() - this asks python to immediately free the memory of any objects that no longer have references

```
import gc as PI_GC

# release intermediate dataframes
B22_02_TT_DF_USR02       = None
B22_03_TT_DF_UST10C_ALL  = None

PI_GC.collect()
```

Despite the above efforts to free-up RAM, we also find that the best way to free RAM is to end the script. This is why it is good practice to have short python scripts for one task only. Each time a python script ends, the RAM that it was using will be released.

Log progress

Log the progress of your script as it runs

If we have heavy data sets our script can take a long time to run. We may wish to log the progress to a log file as the script is running. We can create a logger. In a separate script, called FC_LOGGER, we create:

```
from dotenv import load_dotenv as PI_LOAD_DOTENV
from pathlib import Path as PI_PATH
import os as PI_OS
import logging as PI_LOGGING

PI_LOAD_DOTENV(override=True)

ZV_ST_LOG_FOLDER  = PI_OS.getenv('ZV_ST_ETL_FOLDER')
ZV_ST_LOG_MODE    = PI_OS.getenv('ZV_ST_LOG_MODE', 'w')
ZV_OB_LOG_FILEPATH = PI_PATH(ZV_ST_LOG_FOLDER) / 'app.log'

ZV_OB_FILEHANDLER = PI_LOGGING.FileHandler(
    filename = ZV_OB_LOG_FILEPATH,
    mode     = 'a' if ZV_ST_LOG_MODE == 'a' else 'w',
    encoding = 'utf-8',
    delay    = False
)
ZV_OB_FILEHANDLER.setLevel(PI_LOGGING.INFO)
ZV_OB_FILEHANDLER.setFormatter(
    PI_LOGGING.Formatter(
        '%(asctime)s | %(levelname)s | %(message)s'
    )
)

ZV_OB_LOGGER = PI_LOGGING.getLogger(__name__)
ZV_OB_LOGGER.setLevel(PI_LOGGING.INFO)
if not ZV_OB_LOGGER.handlers:
    ZV_OB_LOGGER.addHandler(ZV_OB_FILEHANDLER)
```

Assuming the above script is called FC_LOGGER.py inside our Z_SHARED_FUNCTIONS folder, we can import the logger and then use it to record the progress in the .log file.

```
from Z_SHARED_FUNCTIONS.FC_LOGGER import ZV_OB_LOGGER

ZV_OB_LOGGER.info(
    f'script started at {PI_DATETIME.now().strftime("%Y-%m-%d %H:%M:%S")}.'
)
```

Useful libraries

Useful libraries (1/2)

Polars

Handle datasets very fast:
- to_Pandas() : convert from Polars to Pandas dataframe
- read_csv() : read a csv file

Pandas

Handle datasets:
- to_excel() : export to Excel

datetime

Get today´s date:
- from datetime import date as PI_DATE
- PI_DATE.today()

json

Import a json file to a dictionary:
- PI_JSON.load() .. Creates a dictionary out of a file object

rapidfuzz

Fast drop-in replacement for the levenshtein library:
from rapidfuzz.distance import Levenshtein as PI_LEVENSHTEIN. rapidfuzz can also be used to do more advanced scoring for fuzzy matching operations.

levenshtein

Compute the distance between two strings in terms of number of different characters.

requests

Download information from a website.

re

Build and apply regular expressions.
- escape() safely escapes user input so that special characters are treated as literals.
- match() applies a regex pattern to a string.

yfinance

Download historical FX rates and stock prices from Yahoo Finance. Useful for revaluing foreign-currency postings.

Useful libraries (2/2)

os

Interact with the Operating System:
- path.join() : create operating system independent file path
- makedirs() : make directory, in case it does not exist

dotenv

- load_dotenv: Import secret variables from a .env file to RAM
- dotenv_values: Import all variables from an end-user variables file

pathlib

- Create paths that are not sensitive to differences between linux and Windows.

logging

- Log the progress of your python script

gc

Garbage collection.
- collect() asks python to free memory immediately after dropping references to large dataframes.

psutil

Check the RAM used by the python script

Working with GitHub

1/ Create an account in GitHub

In case you don't have account, click here to create an account: **https://github.com/**

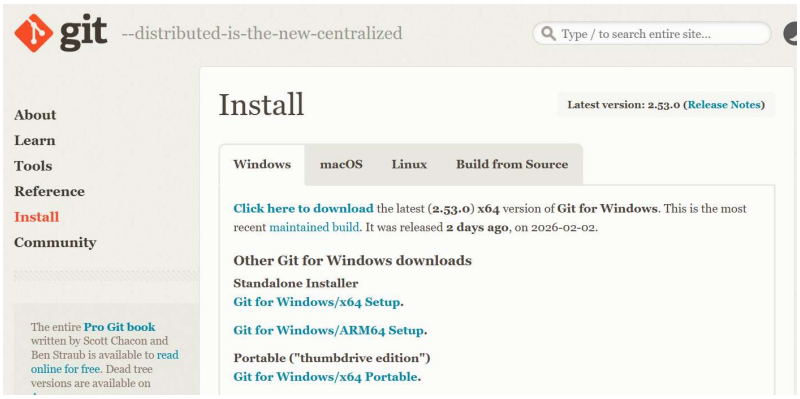
Owners of GitHub repositories will then be able to add you as members.

Note: You will need to do this step before being able to download repositories to your machine.

2/ Download and install the git application

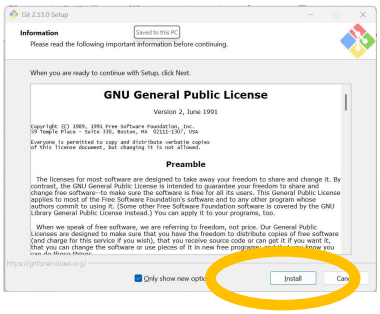
If you would like to be able to clone projects from GitHub, it is necessary to install git on the machine where you use Visual Studio Code.

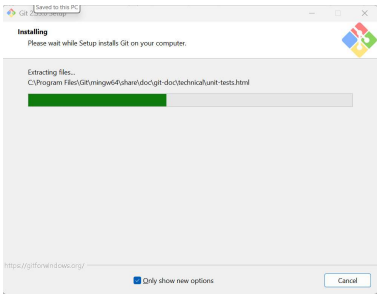
Download the git application from this link and install it on your machine: **https://git-scm.com/download/win**

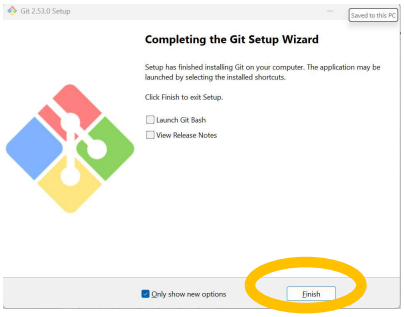


The following installer will be downloaded to your machine: (Double-click on the .exe file to launch it.)

Click Install, choose Next for all questions, then at the end, un-tick both boxes and click Finish.

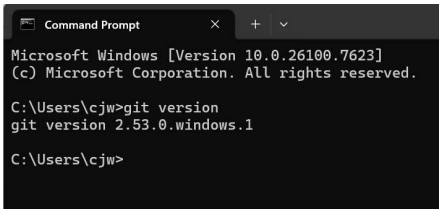






To test that you have correctly installed the git application, type CMD in the Windows Search bar, and then in the black box, type:

```
git version
```



If the git application is correctly installed, you will see the version number.

3/ Configure git application

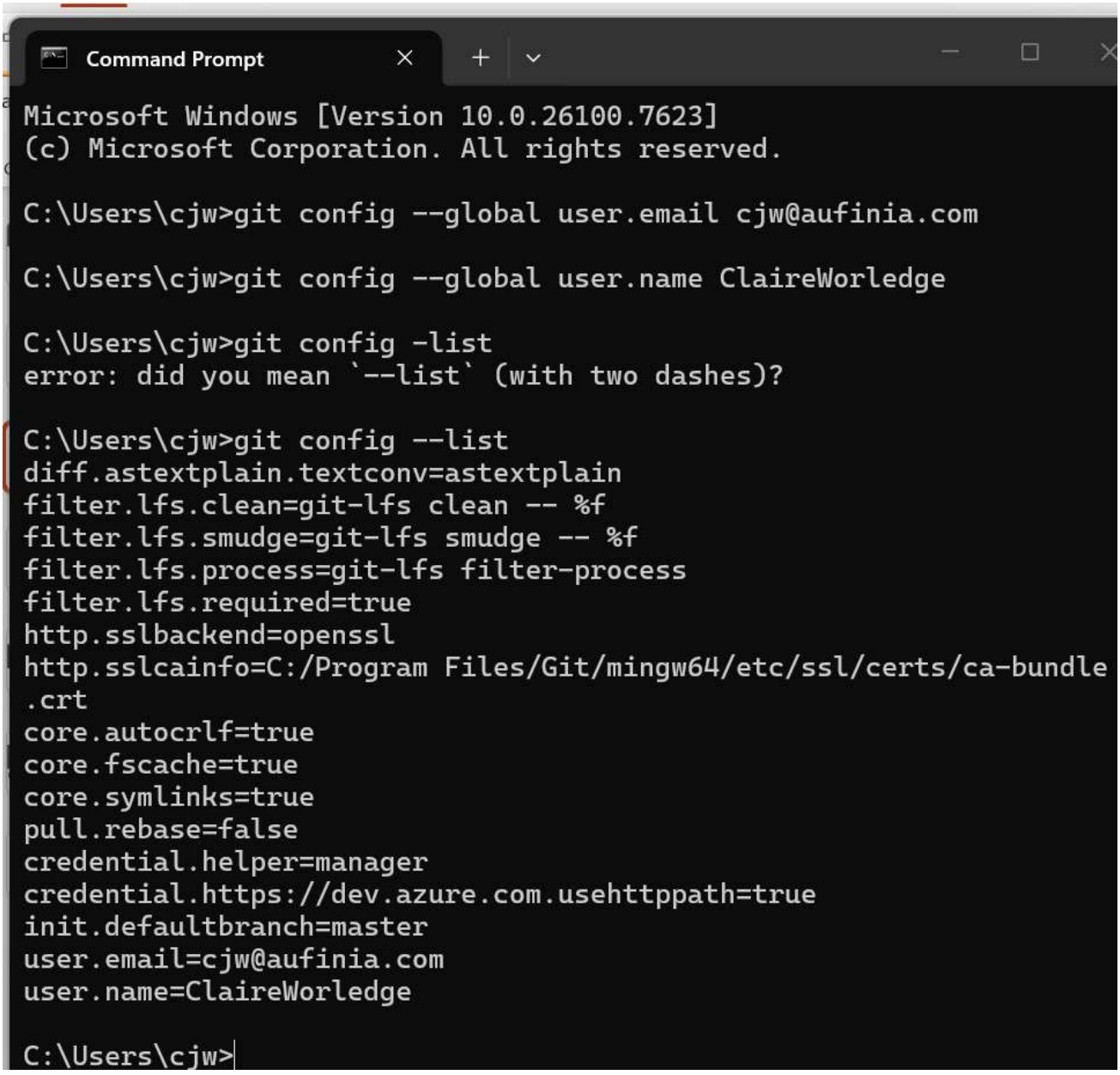
Next, we need to configure our git application, so that it can login to the GitHub account that you created in step 1.

- Type CMD in the search bar of Windows
- Then, in the black box type the following, replacing <your email> and <your name> with the email and username for your GitHub account:

```
git config --global user.email <your email>
git config -- global user.name <your name>
```

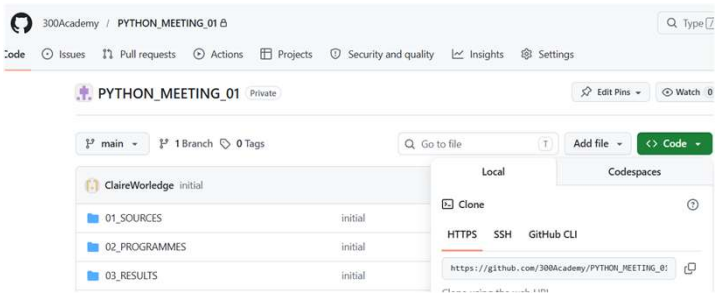
- To test that your email and username were configured correctly, type:

```
git config --list
```



4/ Clone the project from GitHub (1/3)

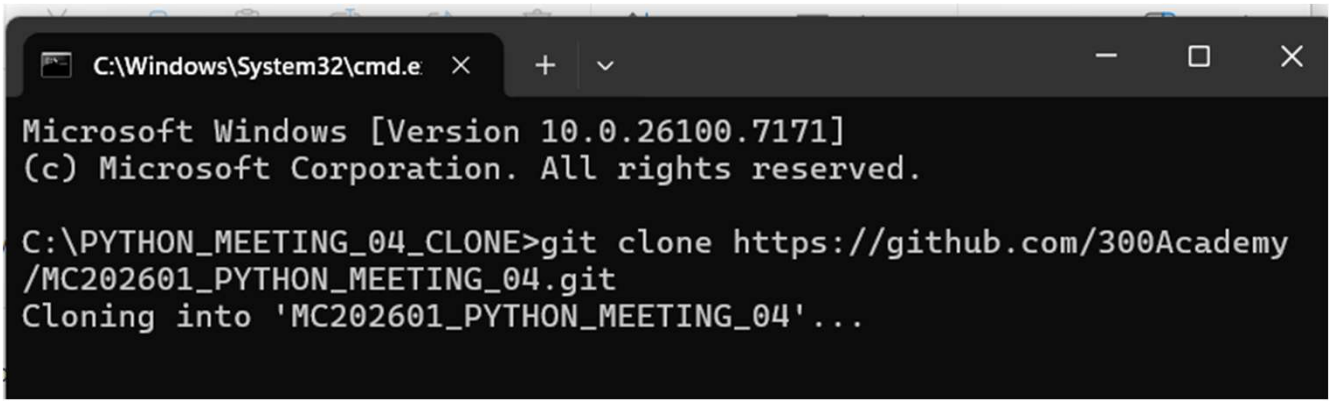
- To clone a GitHub repository, we need to get the repository link. We can get the repository link by going to the GitHub repository and clicking on the button code. You can then copy the link.



- Go to the Windows folder, where you would like to put the python project that you will clone.
- In the Windows search bar of that folder, type CMD
- In the black box that then appears, type the following:

```
git clone <repository link>
```

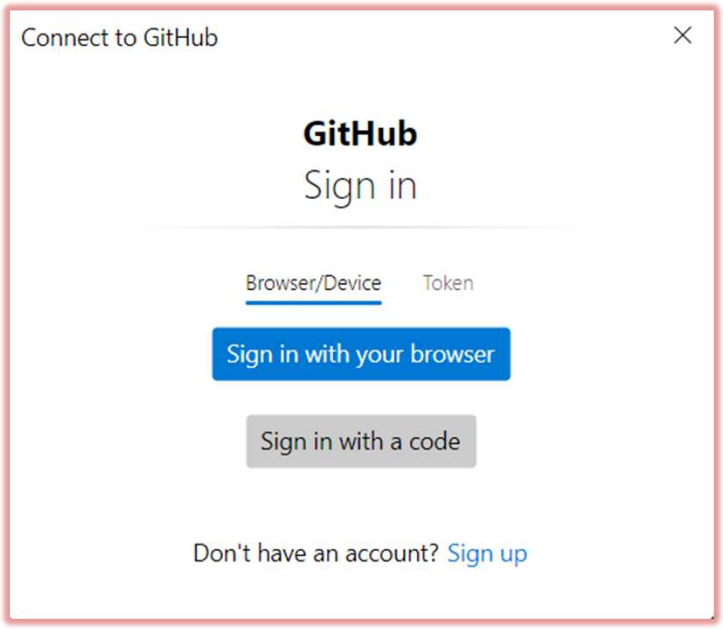
- You will see this message:



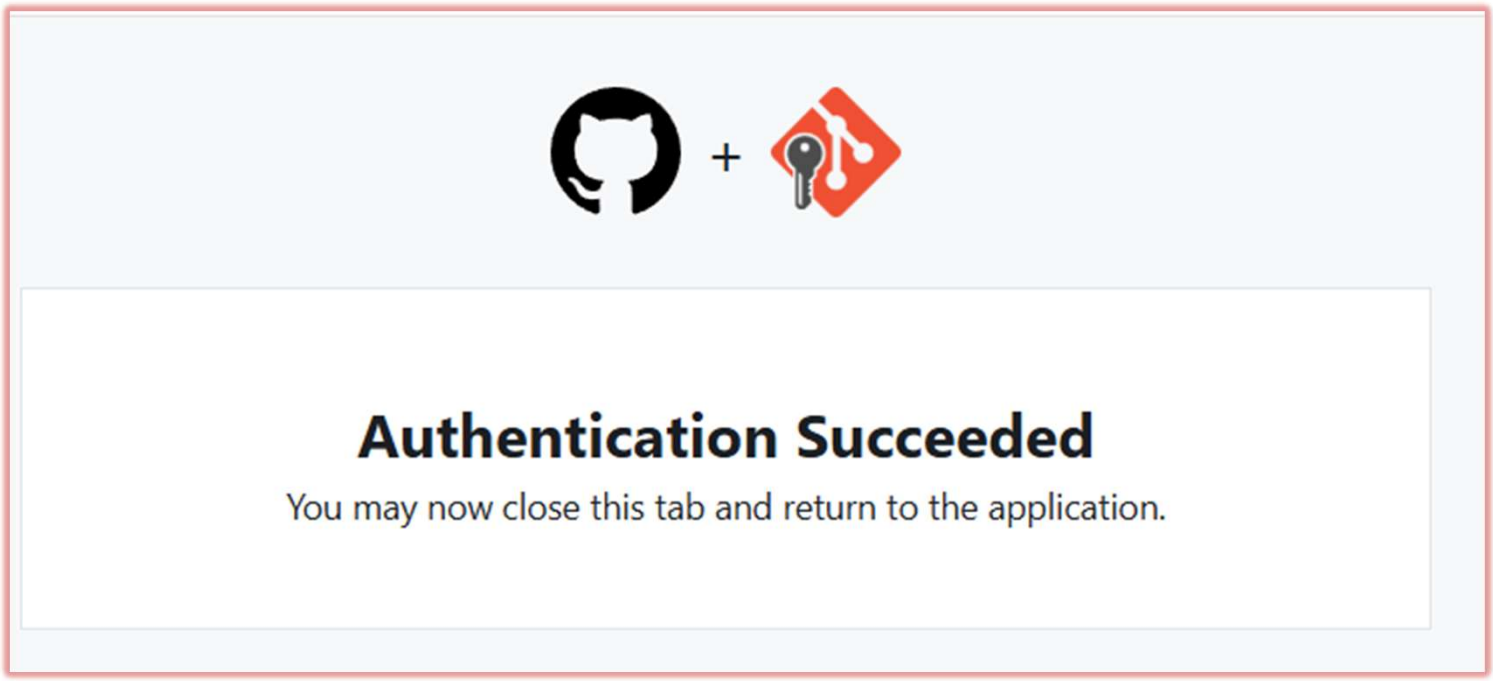
- At the same time, you will see a pop-up, as below. Click on the pop-up icon to sign in to GitHub: (Note, if you are stuck at this point, it might be because you are signed in to a different GitHub account on your machine – see the trouble shooting section).

4/ Clone the project from GitHub (2/3)

- When you click on the little icon from the previous page, you will see this Window. Here you can sign in to GitHub.

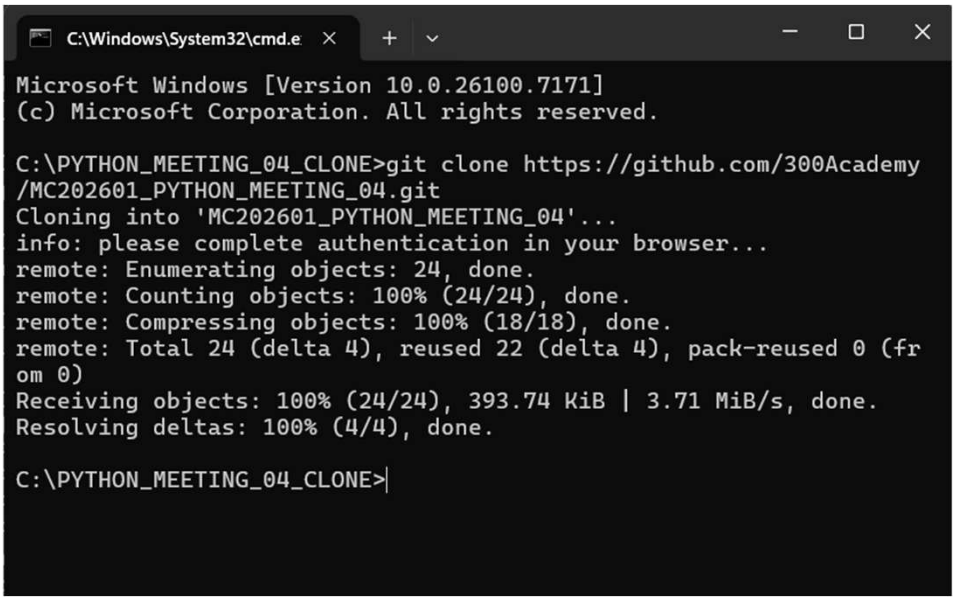


- If the sign in is successful, then you will see this window:

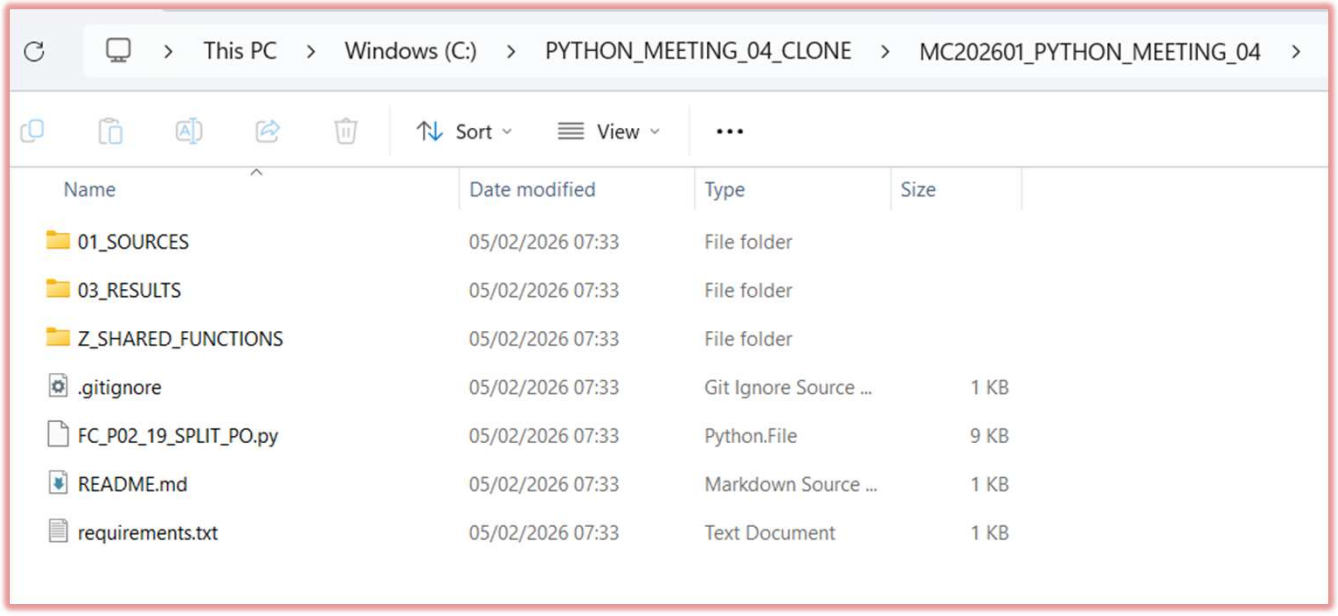


4/ Clone the project from GitHub (3/3)

- Once you have signed-in successfully to GitHub, the folder will be downloaded to the location that you chose before:



- You should then be able to see the folder structure and content in Windows:



5/ Run the code that you downloaded

When the project is downloaded from GitHub:

- Go to the 02_PROGRAMMES folder in Windows
- Type CMD in the Windows search bar and type:

```
code .
```

- Check the .toml to get the python version.
- Create the virtual environment with the correct python version:

```
python -<python version in .toml file> –m venv .venv
```

- Activate the venv:

```
.\venv\Scripts\Activate.ps1
```

- Install libraries:

```
pip intall –r requirements.txt
```

- Update any variables that are in the AM_VARIABLES.txt file:

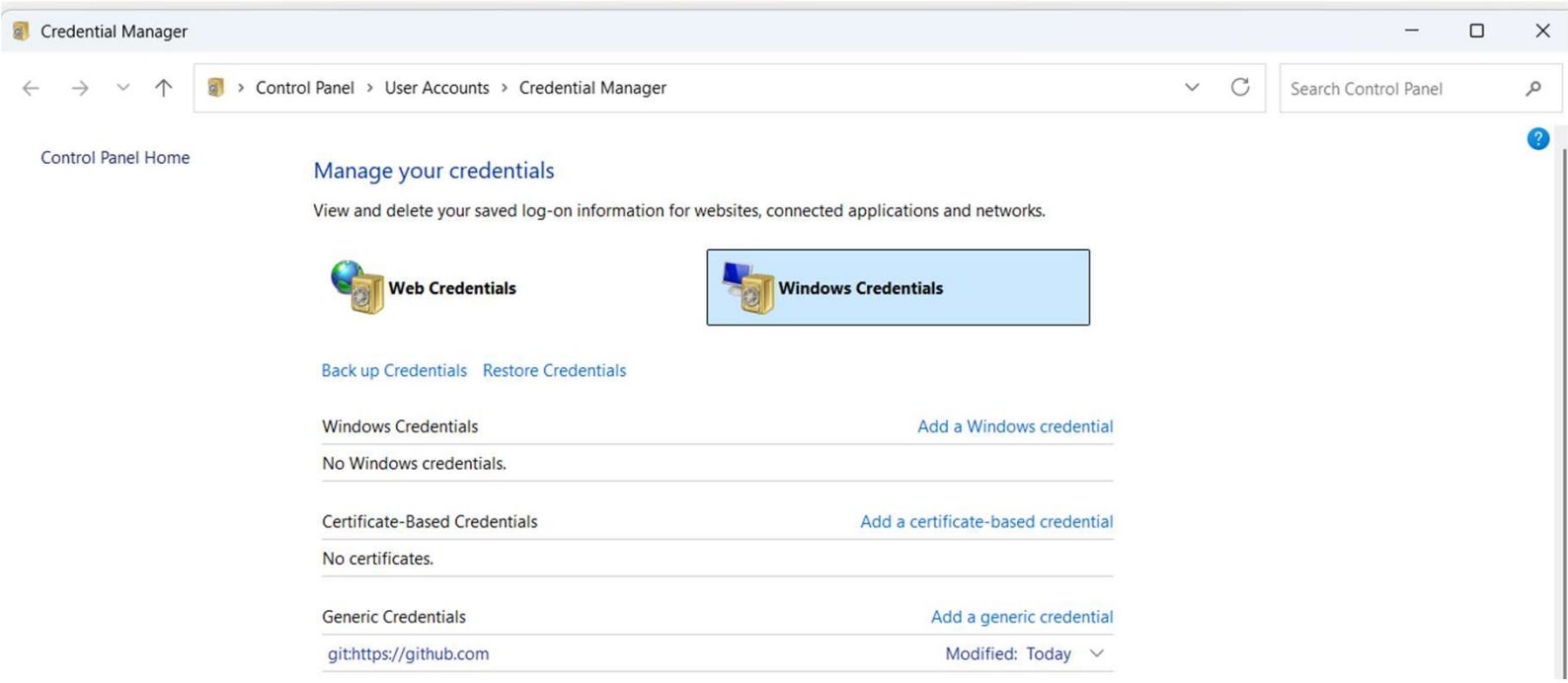
```
AnalysisStartDate=20220101
AnalysisEndDate=20231231
Language='E, EN'
MaximumRAM=20
MaximumCPU=5
```

- Run the program: we can now run our python script, by opening the main script in the project and clicking on the play button at the top-right of Visual Studio Code:

6/ Trouble-shooting cloning from GitHub

- If you are not able to clone the GitHub library, it could be that you already have a GitHub account configured for your machine. In this case, you may need to remove that GitHub account (but check with your IT department first).

- To remove the GitHub account, you can type Credential Manager.
- You will then see the Credential Manager box pop-up:



- You could try removing the GitHub account from Web credentials and starting again the process from step 1.

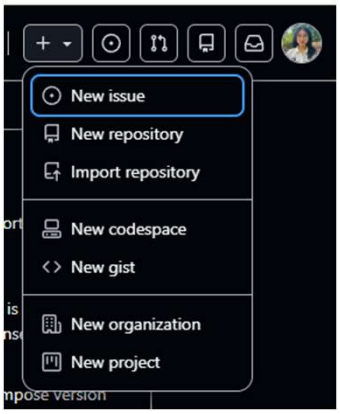
7/ Create your own organization in GitHub

- Note: never store any keys or passwords in code that is uploaded to GitHub. If you store keys, such as ChatGPT keys or the like, then it is highly likely that your code will be scrapped (even if it is private) and your keys compromised. This situation can cause you unexpected bills for the use of services, such as ChatGPT, Azure or AWS, etc.

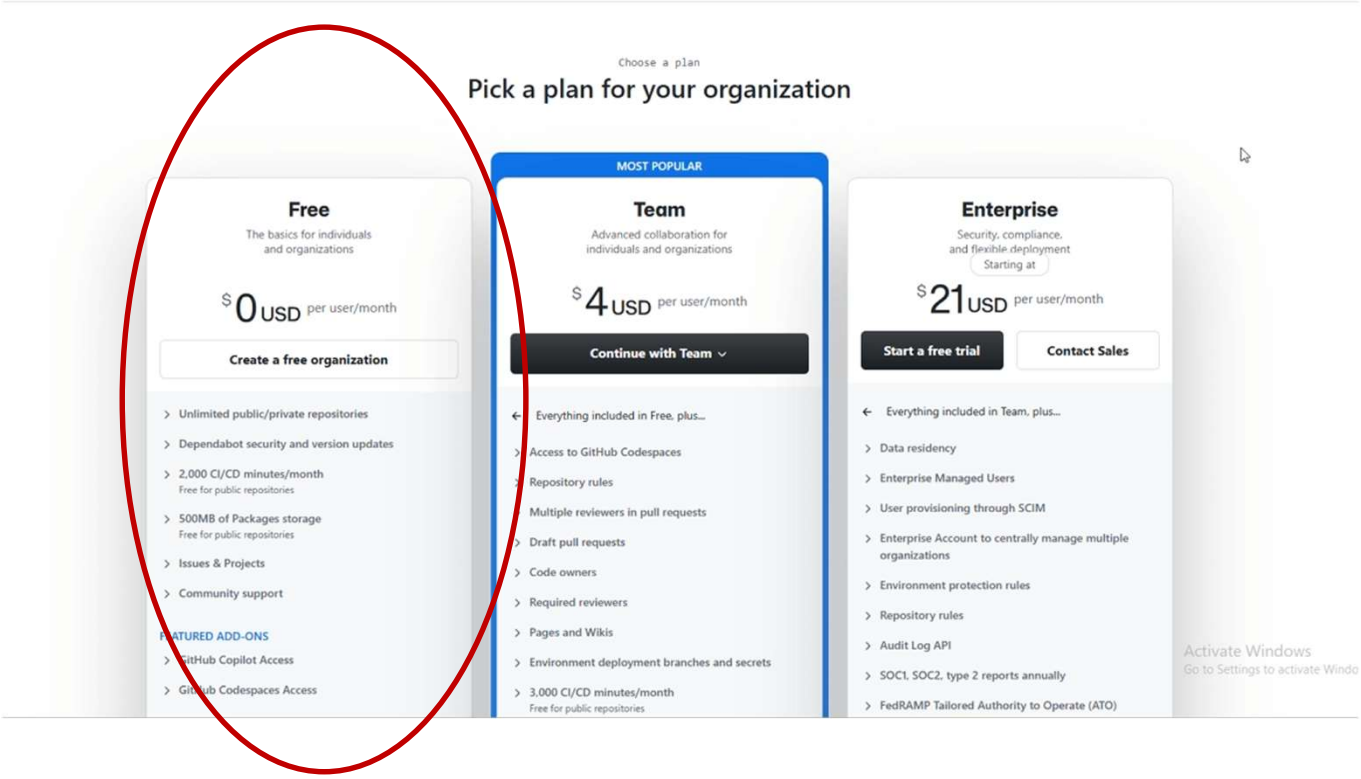
- For this reason, it is important that only authorized team members be allowed to upload code to your GitHub.

- If you want to be able to store your projects in your own GitHub, or share them with your colleagues, then you could create your own organization

- Go to **https://github.com/** to login
- Click on button '+' → choose New organization.

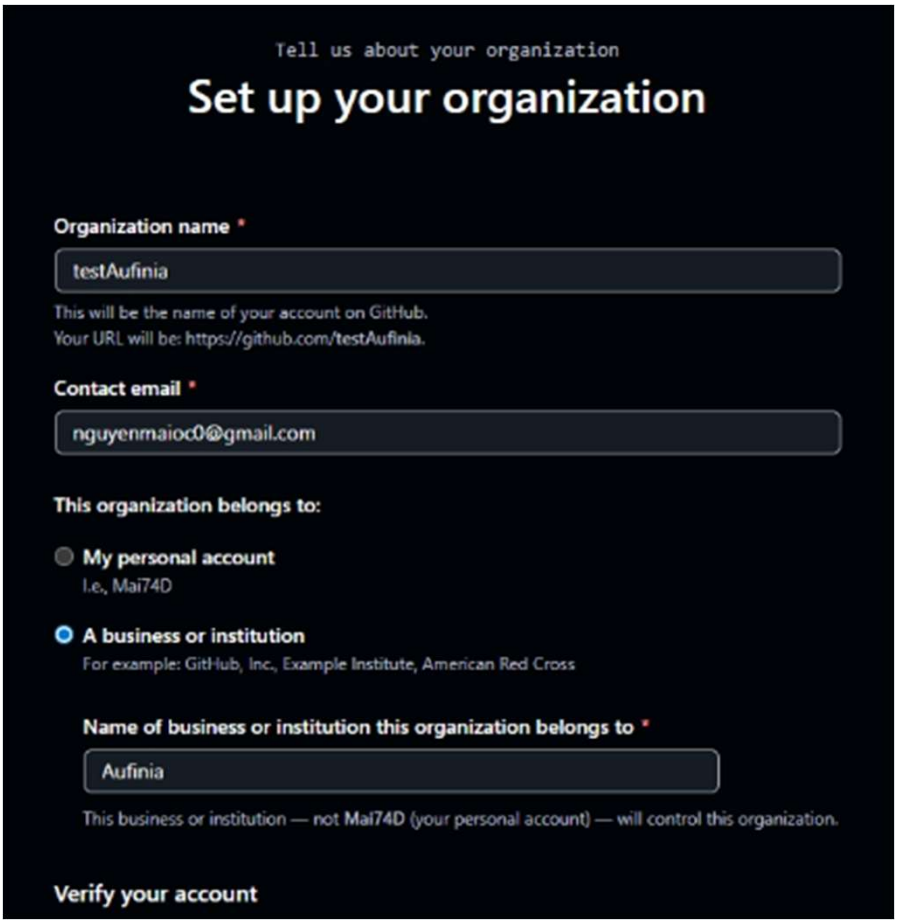


- Choose a plan for your organization: (you can probably use the free plan for small projects):

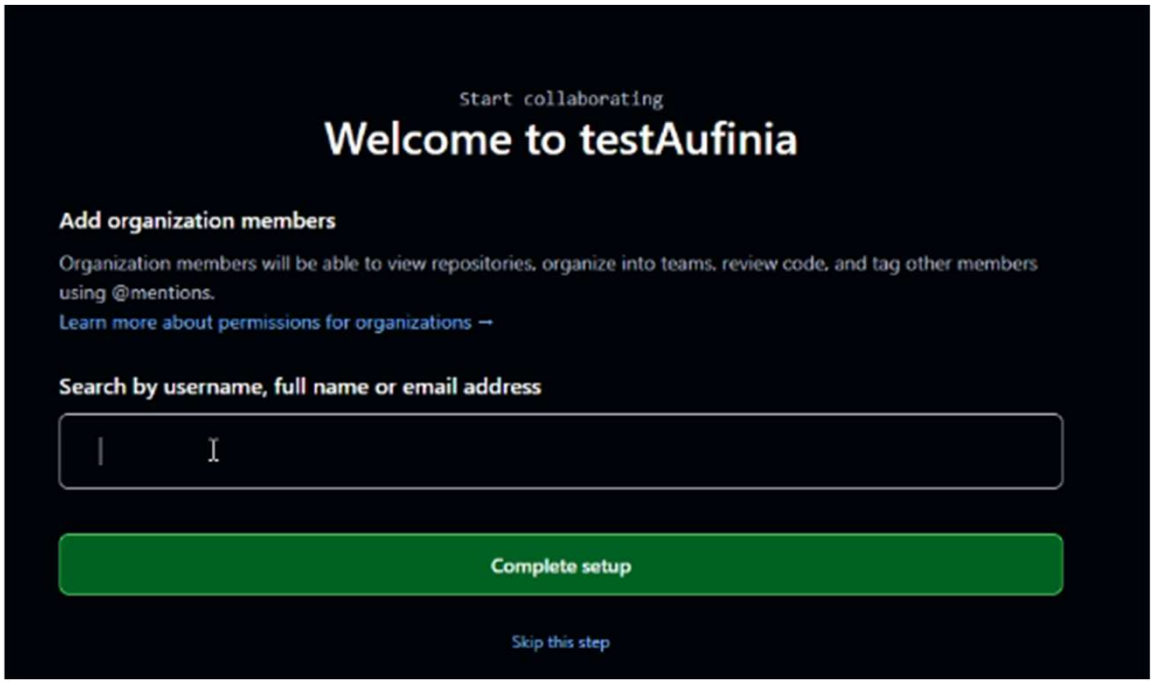


8/ Add name and team members to your organization

- Add information about your organization

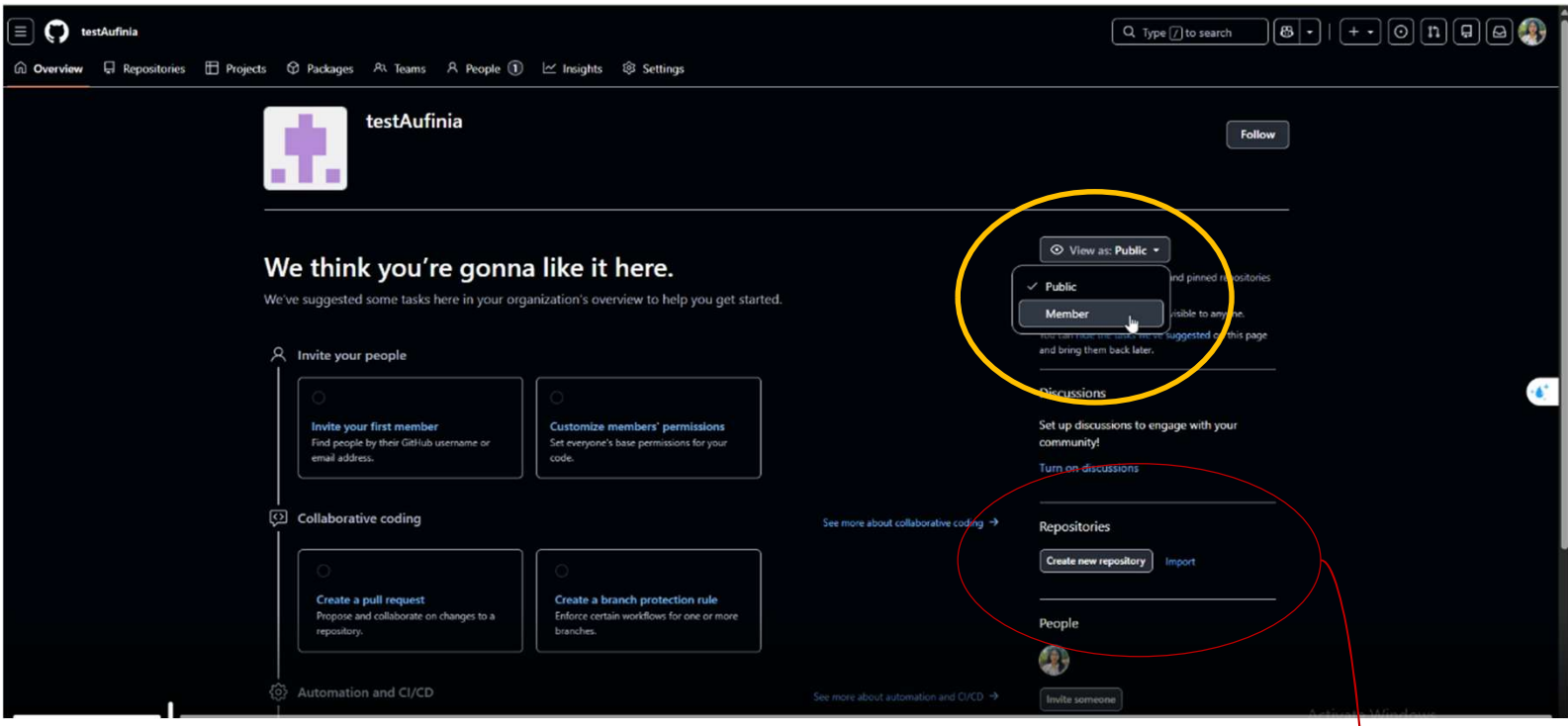


- Add members (optional)

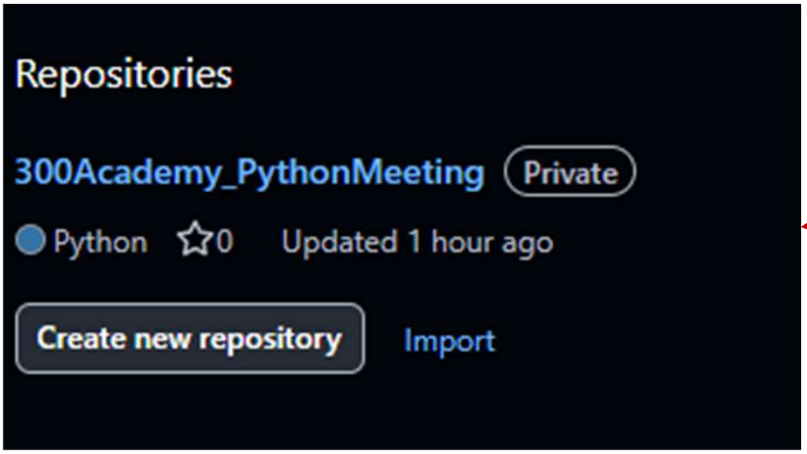


9/ Create a repository in your organization (1/3)

- You can view your organization as the public will see it or as members will see it. Note: this setting is only for the organization page, it does not impact the security settings of the repositories.

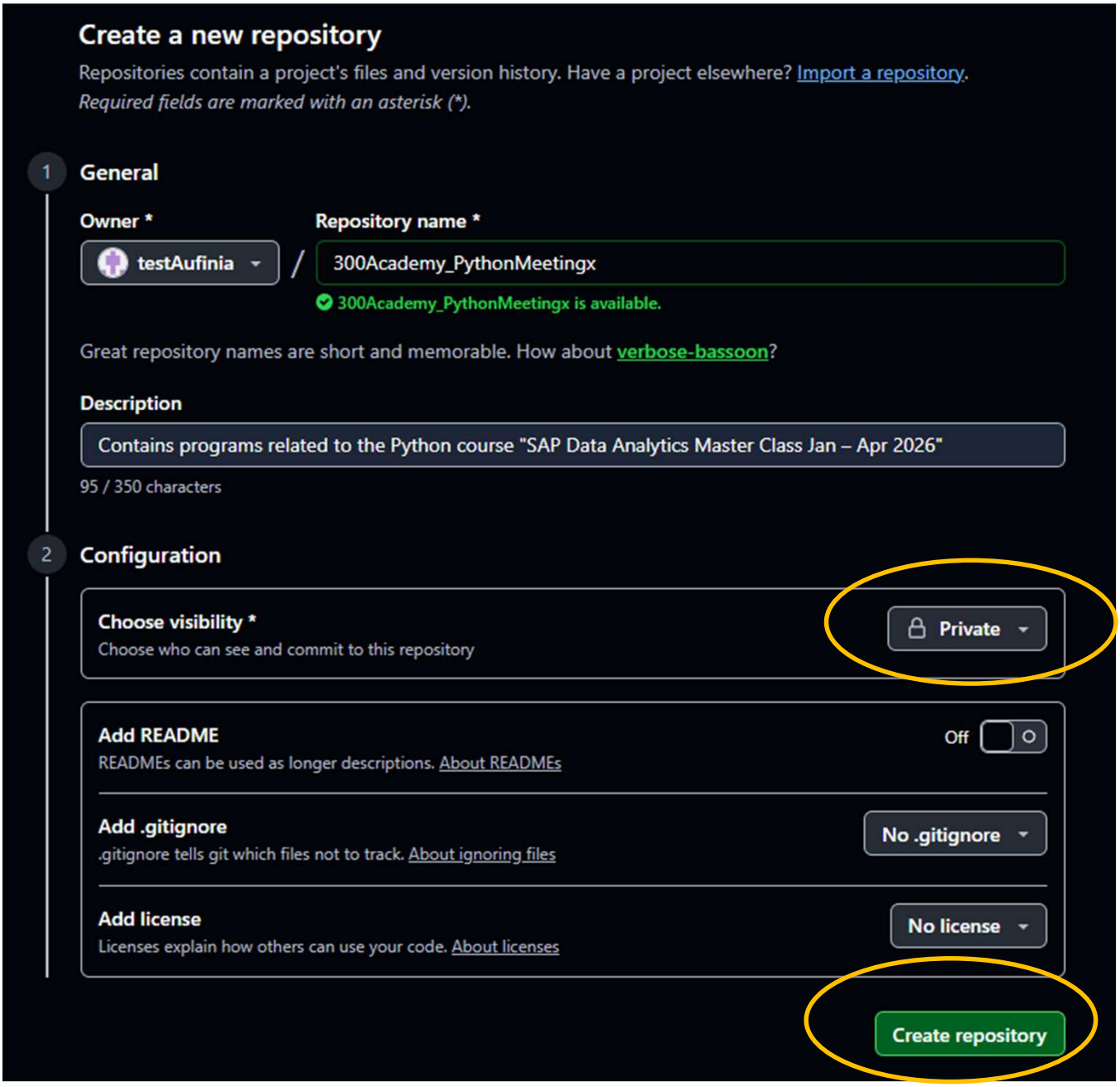


- To add a project (for example a python meeting project), click on Create new repository



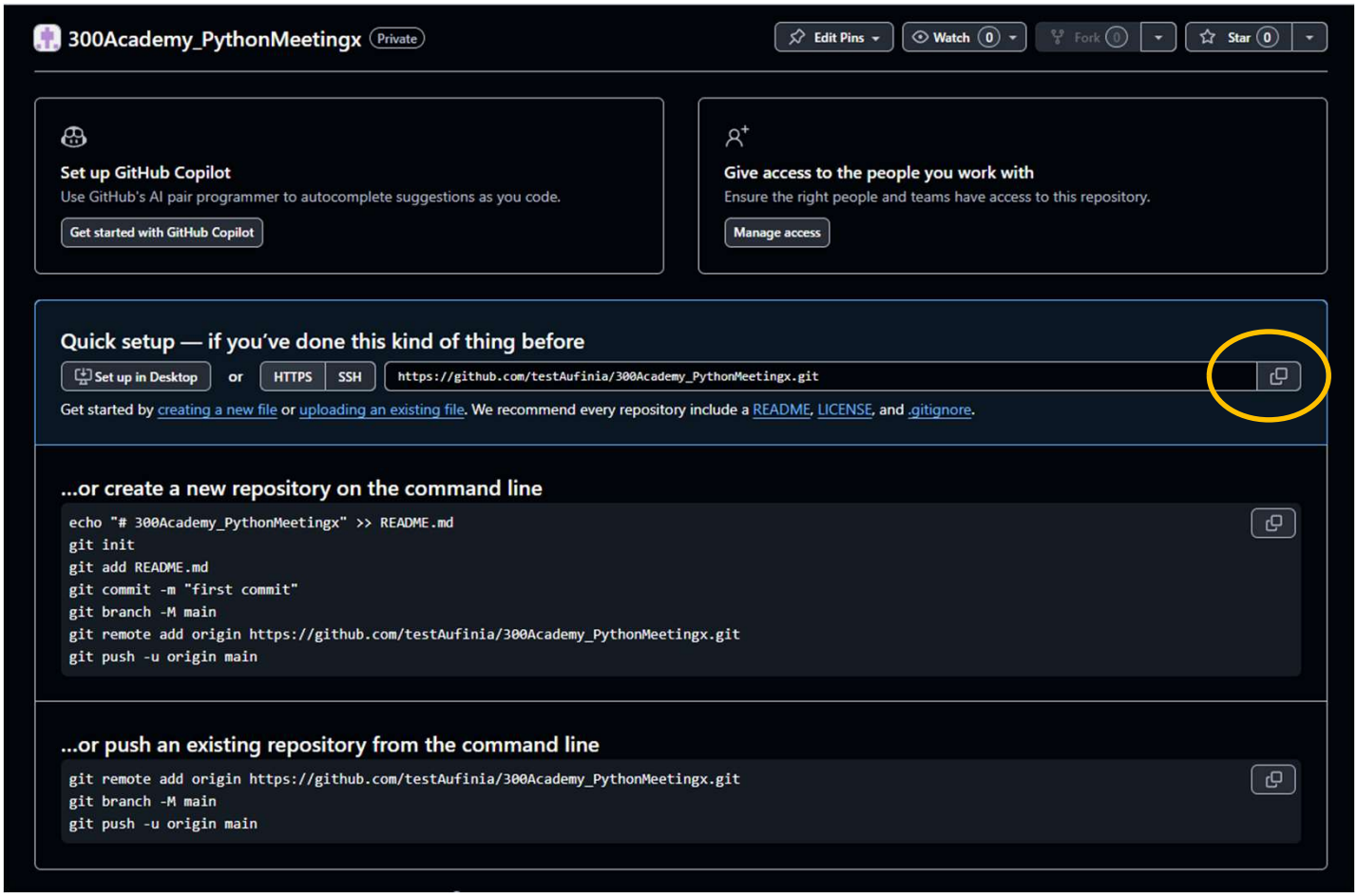
9/ Create a repository in your organization (2/3)

- Fill in the repository name and set to private if you only want your repository members to see this repository.



9/ Create a repository in your organization (3/3)

- Once you have finished creating the repository you will see the below screen.
- Click to copy the link for the next step.



10/ Erase a token

Before being able to push to the GitHub, you need to generate a token. If you previously generated a token, and this token has now expired, you should erase the token before creating a new one.

In the Windows address bar, at the root of your project, type CMD and type:

```
code .
```

Visual Studio Code should open at the root of your project. Go to View-> Terminal and type the following:

- Find out if your token is of type manager or manager-core. If it is, you will be able to erase it in the next step.

```
git config --get credential.helper
```

- Erase the token:

```
@"
protocol=s
host=GitHub.com

"@ | git credential-manager erase
```

11/ Create a token

Before being able to push to the GitHub, you need to generate a token.

- Login to your GitHub account and then got to: **https://github.com/settings/tokens/**
- Click generate new token
- Choose Classic
- Give your token a name in Note
- Choose the token expiration (for example 90 days)
- Tick the scope: repo
- Click Generate token
- Copy the token and remember it.. Note: the token will be used when we do the push of our code to GitHub.

12/ Clone your local project to your GitHub repository (1/2)

Assuming that you have installed the git application on your machine (as mentioned in steps 2 and 3), we can now use git to copy your project to your new GitHub repository.

In the Windows address bar, at the root of your project, type CMD and the:

```
code .
```

Visual Studio Code should open at the root of your project. Go to View-> Terminal and type the following:

- Create a hidden git folder inside your project root folder. Note: if you want to see this hidden folder go to any folder in Windows and select View->Show->Hidden items.

```
git init
```

- Tell git to ignore certain folders and files. For exemple, we definitely want to ignore .venv and .env. There may be other types of files that we want to ignore:

```
@"
*.pptx
*.pdf
*.log
.env
__pycache__/
venv/
*.pyc
.vscode/
.idea/
dist/
build/
*.sav
"@ | Set-Content .gitignore
```

- (Optional): To review what you have in gitignore, you can type:

```
code .gitignore
```

12/ Clone your local project to your GitHub repository (2/2)

- Stage all content inside the root folder, except content that is mentioned in gitignore. Note: when using git, content is staged, before it is committed (copied to GitHub).

```
git add .
```

- (Optional) If you made a mistake and staged the wrong files, you can un-stage:

```
git restore -staged .
```

- To add a message to the commit, type the following:

```
git commit -m “initial commit”
```

- Tell git where the staged project should go:

```
git remote add origin <Repository link that you obtained by clicking on the code button in your new repository>
```

- To create a main branch in the repositor: (Note: -M is used to force, even if the branch already exists or has a different name):

```
git branch -M main
```

- To finally send the staged project (origin) to your GitHub repository (main): (Note: the -u here adds tracking. This means that next time you can just write git push, instead of git push -M main.) (Note also: The first time you do a push after creating a new GitHub token, a pop-up window will appear, requesting authentication. Choose Token and enter the token that you created in step 10.)

```
git push -u origin main
```

13/ Update changes to your GitHub repository

If you change something in your project, follow the following steps to update the files in your GitHub repository:

- Go to the root of your project

```
cd ..
```

- or

```
cd path\to\your\project
```

- Check what you changed: Note: this is really useful for seeing a summary of changes since the last commit:

```
git status
```

- Stage everything that was changed

```
git add .
```

- (Optional): you can specify specific files:

```
git add file1.py file2.py
```

- commit the changes

```
git commit -m “Description of changes”
```

- Send the updated files to GitHub. GitHub will put the new versions and keep the history of the old.

```
git push
```