Running the CDISC Open Rules Engine (CORE) in BASE SAS©

Lex Jansen, CDISC, USA

# Abstract

CDISC Conformance Rules are an integral part of the Foundational Standards and serve as the specific guidance to Industry for the correct implementation of the Standards in clinical studies. The overall goal of the CORE Initiative is to provide a governed set of unambiguous and executable Conformance Rules for each Foundational Standard, and to provide an open-source execution engine for the executable Rules which are available from the CDISC Library [1][2]. The source code of the CORE engine is available on the GitHub repository. A Command Line Interface (CLI) is available on the repository which allows users to run the rules under Windows, Mac, and Linux. If users want to run the Engine in their own Python environment or tooling, it can be implemented as it is available on PyPi (Python Package Index).

For SAS users it is not always an option to run applications as a Command Line Interface.

The presentation will begin with a brief overview of CDISC CORE. The CORE Engine will then be covered. Then the presentation will describe a proof of concept where the CDISC CORE CLI commands have been implemented into SAS processes as Python functions in PROC FCMP, passing parameters and code to the Python interpreter and returning the results to SAS. These Python functions can be called and executed by user-defined SAS functions, which can be called from the DATA step or any context where SAS functions are available.

# Introduction

CDISC Conformance Rules are an integral part of the CDISC Foundational Standards and serve as the specific guidance to Industry for the correct implementation of the Standards in clinical studies. The overall goal of the CORE (CDISC Open Rules Engine) Project is to deliver a governed set of unambiguous and executable Conformance Rules for each Foundational Standard, and to provide a reference implementation of an open-source execution engine for the executable Rules which are retrieved from the CDISC Library.

All code used in this paper and the latest version of the paper are available on GitHub: <https://github.com/lexjansen/cdisc-core-sas>. When encountering issues with the code, please open an issue at <https://github.com/lexjansen/cdisc-core-sas/issues>.

# CORE Concept

CORE consists of two parts:

* The Conformance Rules (“Rules”)
* The Conformance Rules Engine (“Engine”).

Figure 1 illustrates the CORE concept. CDISC, with the help of the CDISC Community, develops the rules according to the process that governs all CDISC Standards development. The Conformance Rules are stored in the CDISC Library along with the rest of the CDISC standards.

The Engine is an open-source software application whose purpose is to execute the Rules against clinical data and return results. The Engine is made available to the CDISC Community in GitHub and users may deploy it in a variety of processing environments including cloud, on-premises desktop, and on-premises server [3]. The Engine accesses the Rules from the CDISC Library via a Library API when it executes. Users may also add custom Rules for processing. The Engine is written in the Python programming language and comes with a permissive MIT open-source license.

Figure 1 The CORE Concept

A diagram of a computer system

Description automatically generated

# CORE Conformance Rules

In the Rule development process human-readable Rule specifications are expressed in a machine-readable form. The human-readable Rule specification is interpreted by the Rule developer and authored in the CORE Rule Editor using a structured language (YAML). The CORE development team has developed a schema that defines the specific YAML syntax for expressing a Rule in YAML [4].

Just like the CORE Engine, the Rule Editor is open-source software that CDISC has made available for free to the CDISC Community via GitHub and which is listed in the CDISC Open Source Alliance (COSA). It has been released under the permissive MIT open-source license [5].

Figure 2 shows an example of a Rule in YAML. In this case the rule states that required variables (Core = “Req”) must be included in the dataset and cannot be null (lines 46-52). The rule is applicable to 3 SDTMIG versions (lines 7-45).The rule is applicable to all domains and classes (lines 66-72). Line 64 shows the message given: At least one Required variable has a null value.

More information about the CORE rule development process and the governance model for the rules can be found in references [2] and [6].

Figure 2 An Example CORE Rule in YAML

A screenshot of a computer program

Description automatically generated

# The CORE Engine

There are several ways to run the CORE Engine:

* As a CLI (Command Line Interface), which allows you to run the rules under Windows, Mac, and Linux. The compiled package can be downloaded, unzipped, and run [7].  
  The repository also contains the steps for creating an executable version.
* Clone the repository and run python core.py from the root of the CORE project with appropriate parameters. See python core.py --help to see the full list of commands [3].
* An alternative to running the validation from the command line is to instead import the rules engine library in Python (available as a package on PyPi) and run rules against data directly (without needing your data to be in .xpt format) in your own environment or tooling [8].

This paper is based on version v.6.3, specifically the Windows CLI (core-windows.zip, October 12, 2023).

The last 2 ways of running - using the non-compiled version – require cloning the cdisc-rules-engine GitHub repository and installing dependencies:

* Clone the repository:

git clone https://github.com/cdisc-org/cdisc-rules-engine.git

* Create a virtual environment:

python -m venv <virtual\_environment\_name>

* Activate the virtual environment:

.\<virtual\_environment\_name>\Scripts\Activate -- on windows

* Install the requirements.

python -m pip install -r requirements.txt

## Running the CORE Engine as a CLI

After the CORE Command has been downloaded and unzipped users can run CORE from a command prompt. Examples of commands will only be given for the Windows operating system. The commands are very similar for other distributions.

To see all available commands run:

.\core –help

Figure 3 shows the available CORE commands.

Figure 3 Available CORE CLI commands

A screenshot of a computer program

Description automatically generated

### The CORE Cache

The CORE Engine stores rules and standards metadata from the CDISC Library as Python pickle files in a local cache folder. Pickle files typically have the . pck extension.

“Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling” or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”. [9].

Figure 4 shows an example of a local CORE cache folder on Windows.

Figure 4 CORE local cache folder with pickle files

A screenshot of a computer screen

Description automatically generated

Rules get added to the CDISC Library on a regular basis. At any moment in time, the locally stored cache can be updated with the update-cache command (see Figure 5) to get the latest set of rules from the CDISC Library. Accessing the CDISC Library requires an API key, which is recommended to define as an environment variable - CDISC\_LIBRARY\_API\_KEY.

To obtain an API key, please follow the instructions found on the CDISC Wiki [10]. Please note it can take up to an hour after signing up to have an API key issued

Figure 5 Update the CORE local cache folder with the update-cache command

A screenshot of a computer program

Description automatically generated

By default, the update-cache command gets the API key from an environment variable (CDISC\_LIBRARY\_API\_KEY) and the default value for the cache path is: ./resources/cache.  
However, both can be specified as parameters (Figure 6).

Figure 6 Updating the CORE local cache folder with the update-cache command

A screenshot of a computer screen

Description automatically generated

Warnings like the following can be ignored:

[WARNING 2024-03-22 15:01:08,502 - connectionpool.py:322] - Connection pool is full, discarding connection: api.library.cdisc.org. Connection pool size: 10

### Validating Data

The following command is used to validate data:

.\core validate -s <standard> -v <standard\_version> -d path/to/datasets

# ex: .\core.exe validate -s sdtmig -v 3-4 -d .\xpt\

Figure 7 shows all available parameters for the validate command (.\core validate –help).

Figure 7 Parameters for the CORE validate command

A screenshot of a computer program

Description automatically generated

The following command will validate the XPT files in folder ..\data\sdtm against the SDTM-IG 3.3 standard using the CDISC/NCI Controlled Terminology package from December 15, 2023:

.\core validate -s sdtmig -v 3-3 -ct sdtmct-2023-12-15 -d ..\testdata\sdtm\ -o sdtmig-3-3-report

The result of this command will be an Excel spreadsheet called **sdtmig-3-3-report.xlsx**. Figure 8 shows some screenshots of this spreadsheet.

Figure 8 Validation Report in Excel

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The Rules Report tab displays the run status of each rule selected for validation. The possible rule run statuses are:

* SUCCESS - The rule ran, and data was validated against the rule. May or may not produce results.
* SKIPPED - The rule was unable to be run. Usually due to missing required data but could also be cause by rule execution errors.

After cloning the cdisc-rules-engine GitHub repository and installing dependencies the CLI can also run as a Python program. Here is an example:

python core.py validate -s sdtmig -v 3-3 -ct sdtmct-2023-12-15 -d ..\testdata\sdtm\ -o sdtmig-3-3-report

# Running the CORE Engine in SAS

## Running a CLI (Command Line Interface) in SAS

SAS has various techniques to execute operating system commands:

* X statement
* SYSTASK statement
* %SYSEXEC statement
* CALL SYSTEM statement
* SYSTEM function
* FILENAMEC statement with the PIPE option

There are a few SAS options related to executing operating system commands:

* XSYNC - Controls whether an X command or statement executes synchronously or asynchronously.
* XWAIT - Specifies whether you must type EXIT at the DOS prompt before the DOS shell closes.
* XMIN - Specifies opening the application specified in the X command in a minimized state or in the default active state.

The following code builds commands to update the cache and then validates data with the CORE engine:

options noquotelenmax;

options noxwait xsync xmin;

%let core\_exe = \\_Projects\CDISC\_CORE\core\core.exe;

%let project\_folder = /\_github/lexjansen/cdisc-core-sas;

%let core\_cache\_folder = &project\_folder/resources/cache;

%let core\_template = &project\_folder/resources/templates/report-template.xlsx;

%let test\_data\_folder = &project\_folder/testdata/sdtm;

%let core\_report = &project\_folder/reports/sdtmig-3-3-report;

%let core\_log = %sysfunc(pathname(work))/core\_command;

%let core\_command\_1 = &core\_exe update-cache;

%let core\_command\_1 = &core\_command\_1 -c &core\_cache\_folder;

%let core\_command\_2 = &core\_exe validate;

%let core\_command\_2 = &core\_command\_2 -ca &core\_cache\_folder;

%let core\_command\_2 = &core\_command\_2 -rt &core\_template;

%let core\_command\_2 = &core\_command\_2 -dp &test\_data\_folder/dm.xpt;

%let core\_command\_2 = &core\_command\_2 -dp &test\_data\_folder/ae.xpt;

%let core\_command\_2 = &core\_command\_2 -s sdtmig;

%let core\_command\_2 = &core\_command\_2 -v 3-3;

%let core\_command\_2 = &core\_command\_2 -ct sdtmct-2023-12-15;

%let core\_command\_2 = &core\_command\_2 -o &core\_report;

%let core\_command\_2 = &core\_command\_2 -r CORE-000006 -r CORE-000007   
-r CORE-000012 -r CORE-000013 -r CORE-000019 -r CORE-000266 -r CORE-000356;

x "&core\_command\_1 > ""&core\_log.\_1.log"" 2>&1";

%put &=sysrc;

**data** \_null\_;

infile "&core\_log.\_1.log" truncover;

input;

put \_infile\_;

**run**;

x "&core\_command\_2 > ""&core\_log.\_2.log"" 2>&1";

%put &=sysrc;

**data** \_null\_;

infile "&core\_log.\_2.log" truncover;

input;

put \_infile\_;

**run**;

Note:

* The code only validates two datasets: ae.xpt and dm.xpt
* The code only validates against a limited set of rules, using the -r command line option
* Command output is saved to a file (> ""&core\_log.\_1.log"" 2>&1) and printed to the log.

This approach works well, but there can be an issue. The assumption is that the X command is valid in the current SAS session. This may not be the case especially in shared SAS environments. In certain SAS environments SAS administrators may have specified the **NOXCMD** or **XCMD OFF** system options. When specified, the **NOXCMD** options disables the following:

* PIPE and NAMEPIPE device types in the FILENAME statement
* CALL SYSTEM routine
* X command
* Dynamic Data Exchange (DDE)
* %SYSEXEC macro
* SYSTASK statement
* PIPE and NAMEPIPE device types in the FILENAME function.

Indeed, specifying -NOXCMD at SAS invocation results in the following message in the SAS log:

51 ;

ERROR: Shell escape is not valid in this SAS session.

52

53 x "&core\_command\_1 > ""&core\_log.\_1.log"" 2>&1"

53 ! ;

The remainder of this paper will show how the use of PROC FCMP works around the **NOXCMD** limitation.

The CDISC CORE CLI commands will be implemented into SAS processes as Python functions in PROC FCMP, passing parameters and code to the Python interpreter and returning the results to SAS. These Python functions can be called and executed by user-defined SAS functions, which can be called from the DATA step or any context where SAS functions are available.

## SAS PROC FCMP with Python Support

Starting with the May 2019 release of SAS 9.4M6, the PROC FCMP procedure added support for submitting and executing functions written in Python from within a SAS session using the new Python object. FCMP, or the SAS Function Compiler, enables users to write their own functions and subroutines that can then be called from just about anywhere a SAS function can be used in SAS. Users are not restricted to using Python only inside a PROC FCMP statement. You can create an FCMP function that calls Python code, and then call that FCMP function from the DATA step [11][12].

### Prerequisites

Before you can run Python code in SAS, you must install SAS 9.4M6 (May 2019 update) or later deployments. The following environment setup steps must be completed before you can use PROC FCMP to run the Python code [13].

* Install Python. The CDISC CORE engine requires Python 3.9 or Python 3.10.
* Set the **MAS\_M2PATH** environment variable to specify the absolute path to the mas2py.py file included in your SAS installation. The mas2py.py file is used to execute Python code within a Python process that is launched by SAS Micro Analytic Service.
* Set the **MAS\_PYPATH** environment variable to specify the absolute path to the Python executable.

In the SAS example in the GitHub repository (<https://github.com/lexjansen/cdisc-core-sas>) that contains the code used in this paper, the two environmental variables are define in a SAS configuration file:

options set = MAS\_PYPATH = "&project\_folder/.venv/Scripts/python.exe";

options set = MAS\_M2PATH = "%sysget(SASROOT)/tkmas/sasmisc/mas2py.py";

The following (optional) environmental variables can be set at the operating system:

* **MAS\_PYLOG\_FILE** - Will create a local Python Logging file. Default: Filename is overwritten before logging. '+log.txt': will append data to 'log.txt'.
* **MAS\_PYLOG\_LEVEL** - The logging level: {ALL, TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF}. Default: WARN.
* **MAS\_PYOUT\_FILE** - Filename for python process STDOUT / STDERR. Default: Filename is overwritten before logging. '+out.txt': will append data to 'out.txt'.

These files can be useful when debugging the execution of Python functions by PROC FCMP.

To be able to run the code that comes with this paper, the user should clone the cdisc-core-sas GitHub repository (<https://github.com/lexjansen/cdisc-core-sas>), create a virtual Python environment, and then install the additional packages that are needed by the CDISC CORE engine:

• Clone the repository:

git clone https://github.com/cdisc-org/cdisc-rules-engine.git

• Create a virtual environment:

python -m venv <virtual\_environment\_name>

• Install the requirements.

python -m pip install -r requirements.txt

The cdisc-core-sas repository comes bundled with the source code of the v0.6.3 release (October 12, 2023) of the CDISC CORE engine.

Figure 9 shows an example of a cloned cdisc-core-sas repository with a virtual environment (.venv).

Figure 9 Cloned Github repository

A screenshot of a computer

Description automatically generated

The Python function called by SAS needs to know where to find the cdisc-rules-engine library. For this reason we need an operational system environment variable (**CORE\_PATH**) that defines this location of the cdisc-rules-engine library[[1]](#footnote-1).

It is also recommended to create an operational system environment variable (**CDISC\_LIBRARY\_API\_KEY**) that has the API key for accessing the CDISC library.

You may have to work with your SAS administrators to implement these prerequisites.

### PROC FCMP Python Objects

PROC FCMP Python objects enable you to embed and import Python functions into SAS programs. The Python code is not converted to SAS code. Instead, the Python code runs in the Python interpreter of your choice and returns the results to SAS. With a small Python code modification, you can run your Python functions from SAS and easily program in both languages at the same time [14][15].

A typical workflow for using Python objects in PROC FCMP is the following:

* Declare a Python object and a dictionary object
* Insert Python source code into SAS
* Publish Python source code
* Call the Python source code
* Return results from the dictionary

Example:

proc fcmp;

declare object py(python);

submit into py;

def PyProduct(var1, var2):

"Output: MyKey"

newvar = var1 \* var2

return newvar,

endsubmit;

rc = py.publish();

rc = py.call("PyProduct", 5, 10);

MyResult = py.results["MyKey"];

file log;

put MyResult=;

run;

The Python object and all the Python object methods are valid only inside a PROC FCMP statement. For example, attempting to declare a Python object in a DATA step program results in an error. However, it is possible to call Python functions from the DATA step by creating PROC FCMP functions or subroutines that contain Python functions. PROC FCMP functions that contain Python functions are valid in the DATA step and can be called like other functions that are created using PROC FCMP.

Also, by using the INFILE method, we can read external source code from a file into a Python object at parse time.

Example:

We have a file (timesfive.py) with the Python code that defines a Python function:

def TimesFive(PythonArg):

"Output: MyKey"

newvar = PythonArg \* 5

return newvar

We can create a SAS function MyPyFunc that calls this Python function. We can use MyPyFunc in a data step:

**proc** **fcmp** outlib=work.fcmp.pyfuncs;

function MyPyFunc(FCMParg);

declare object py(python);

submit into py("&project\_folder/python/timesfive.py");

rc = py.publish();

rc = py.call("TimesFive",FCMParg);

MyFCMPResult = py.results["MyKey"];

return(MyFCMPResult);

endfunc;

**run**;

options cmplib=work.fcmp;

**data** \_null\_;

x = MyPyFunc(**5**);

put x=;

**run**;

The LOG file will display x=25.

There are a few limitations related to PROC FCMP functions:

* User-defined functions in PROC FCMP only support positional parameters.
* User-defined functions in PROC FCMP do not support optional parameters.
* User-defined functions in PROC FCMP do not support default values for parameters.

Especially with CORE commands that have many options – like the CORE validate command – these limitations are problematic. Users would have to remember the exact order of the parameters and specify values that are common defaults. A solution for these issues is to wrap the functions in macros that can have named parameters and default values.

### Turning CORE CLI Commands into PROC FCMP Functions

We saw in Figure 3 (Available CORE CLI commands) that the CORE CLI supports several commands. We want to be able to run these commands in SAS. To support this, we will:

* Create Python functions where the command options are function parameters
* Create PROC FCMP functions or subroutines that contain the Python functions
* Create macros that call the functions and subroutines. These macros can also do parameter checks that are not already done by the Python functions.

All CORE CLI commands were implemented (Table 1), except for the **test** command. The **test** command is used by CORE Rule developers. The scope for our implementation is the validation of data.

Table 1 CORE CLI commands implementation

|  |  |  |  |
| --- | --- | --- | --- |
| CLI Command | Python Function | PROC FCMP SAS Function | SAS Macro |
| list-ct | core\_list\_ct | core\_list\_ct | core\_list\_ct |
| list-dataset-metadata | core\_list\_dataset\_metadata | core\_list\_dataset\_metadata | core\_list\_dataset\_metadata |
| list-rule-sets | core\_list\_rule\_sets | core\_list\_rule\_sets | core\_list\_rule\_sets |
| list-rules | core\_list\_rules | core\_list\_rules | core\_list\_rules |
| test | - | - | - |
| update-cache | core\_update\_cache | core\_update\_cache | core\_update\_cache |
| validate | core\_validate\_data | core\_validate\_data | core\_validate\_data |
| version | core\_version | core\_version | - |

In the **cdisc-rules-engine** GitHub repository there is a Python file, core.py, that defines the CORE CLI.

The core.py file uses the Click package to create the command line interface [16].

Appendix 1 shows the Python code in core.py to implement the **validate** command. The Python package imports are not displayed.

To turn this code in a Python function that can be used by PROC FCMP we have to do several things:

* Wrap the code in a Python function, for example core\_validate\_data.
* Take out all code related to the Python Click package (@click decorators, ctx function argument).
* Replace ctx.exit() statements with return statements, adding return messages where needed.
* Add default values to the Python validate() function.
* Since SAS does not support Tuples, we need to convert a comma separated list into a Python supported datatype, like tuples and lists.
* Move parameter checks from the Click package to the SAS macro,for example %core\_validate\_data.
* The Python function called by SAS needs to know where to find the cdisc-rules-engine library. For this reason we need an operational system environment variable (CORE\_PATH) that defines this location. Every Python function will have code added to find the cdisc-rules-engine library:  
    
  # Add top-level folder to path so that project folder can be found  
  core\_path = os.environ["CORE\_PATH"]  
  lib\_path = os.path.abspath(os.path.join(\_\_file\_\_, core\_path))  
  if lib\_path not in sys.path: sys.path.append(lib\_path)

Appendix 2 shows the Python function that is called by SAS PROC FCMP implement the **validate** command.

Now that the Python functions have been defined, they can be called by a SAS function in PROC FCMP. Below is the code used for the **core\_version**, **core\_data\_validate**, and **core\_update\_cache** functions:

**proc** **fcmp** outlib = macros.core\_funcs.python;

function core\_version() $ **32**;

length message $ **128**;

declare object py(python);

submit into py("&project\_folder/python/core\_version.py");

rc = py.publish();

rc = py.call('core\_version');

message = py.results['message\_return\_value'];

return(message);

endfunc;

function core\_validate\_data(

cache $, pool\_size, data $, dataset\_path $, log\_level $,  
 report\_template $,standard $, version $, output $,   
 output\_format $, raw\_report, controlled\_terminology\_package $,   
 define\_version $, data\_format $, define\_xml\_path $,whodrug $,   
 meddra $, rules $) $ **128**;

length message $ **128**;

declare object py(python);

submit into py("&project\_folder/python/core\_validate\_data.py");

rc = py.publish();

rc = py.call('core\_validate\_data',

cache, pool\_size, data, dataset\_path, log\_level, report\_template,  
 standard, version, output, output\_format, raw\_report,   
 controlled\_terminology\_package, define\_version, data\_format,   
 define\_xml\_path, whodrug, meddra, rules);

message = py.results['message\_return\_value'];

return(message);

endfunc;

subroutine core\_update\_cache(apikey $, cache\_path $);

declare object py(python);

submit into py("&project\_folder/python/core\_update\_cache.py");

rc = py.publish();

rc = py.call('core\_update\_cache', apikey, cache\_path);

endsub;

...

**run**;

### Creating Macros to Run CORE Commands

The last step is to create macros that call the PROC FCMP functions so that we can define named parameters and default parameter values.

Also, we can do additional parameter validation. Some of the validation checks:

* Does a required parameter have a value?
* Does a file exist?
* Does a folder exist?

Below is an example of a macro, in this case a partial listing of the **core\_validate\_data** SAS macro.

**%macro** core\_validate\_data(

cache\_path = %sysfunc(sysget(CORE\_PATH))/resources/cache,

pool\_size = **10**,

data =,

dataset\_path =,

log\_level = disabled,

report\_template = %sysfunc(sysget(CORE\_PATH))/resources/templates/report-template.xlsx,

standard =,

version =,

controlled\_terminology\_package =,

output =,

output\_format = XLSX,

raw\_report = **0**,

define\_version =,

data\_format = XPT,

define\_xml\_path =,

whodrug =,

meddra =,

rules =

) / minoperator;

...

%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

%\* Parameter checks \*;

%\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*;

...

data \_null\_;

message = core\_validate\_data("&cache\_path", &pool\_size, "&data", "&dataset\_path", "&log\_level", "&report\_template", "&standard", "&version", "&output", "&output\_format", &raw\_report, "&controlled\_terminology\_package" , "&define\_version", "&data\_format", "&define\_xml\_path", "&whodrug", "&meddra", "&rules");

if not missing(message) then putlog "ERR" "OR: " message;

run;

%exit\_macro:

**%mend** core\_validate\_data;

Parameter descriptions can be found in the macro headers.

## Using the Macros to Run CORE Commands

This section gives examples on the use of the macros that implement the CORE commands.

For every macro there is an example program in programs folder in the GitHub repository (<https://github.com/lexjansen/cdisc-core-sas>).

Before running any of these example programs the user needs to run the **create\_core\_functions.sas** program in the **programs** folder to create the PROC FCMP functions dataset.

Every example program starts with the same lines of code:

%\* This code assumes that your SAS environment can run Python objects. ;

%\* Check the programs/config.sas file for Python configuration. ;

%\* update this macro variable to your own location;

%let project\_folder = /\_github/lexjansen/cdisc-core-sas;

%include "&project\_folder/programs/config.sas";

Make sure to update the **project\_folder** macro variable to your own location.

### %core\_update\_cache

Purpose: get the latest set of rules and standards metadata from the CDISC Library.

%***core\_update\_cache***(

/\* apikey= <your API key>, \*/

cache\_path = &project\_folder/resources/cache

);

This macro call assumes that you have an environment variable CDISC\_LIBRARY\_API\_KEY. If not, you can specify the API key in the macro call.

The result of this macro call is that the latest set of rules and standards metadata from the CDISC Library is extracted to the local cache folder (see Figure 4 CORE local cache folder with pickle files).

### %core\_list\_ct

Purpose: list the Controlled Terminology packages available in the cache.

filename ct "&project\_folder/json/core\_ct.json";

%***core\_list\_ct***(

subsets =,

output = %sysfunc(pathname(ct)),

cache\_path = &project\_folder/resources/cache

);

**data** \_null\_;

rc = jsonpp('ct','log');

**run**;

libname jsonfile json fileref=ct;

**data** metadata.core\_ct(keep=value rename=(value=ct\_package));

set jsonfile.alldata;

**run**;

filename ct clear;

libname jsonfile clear;

The macro extracts a JSON file with the available Controlled Terminology packages in the cache. The JSON file can be easily converted to a SAS dataset as the code demonstrates. The dataset has one variable (ct\_package) with the Controlled Terminology package.

Figure 10 core\_ct dataset with available Controlled Terminology packages

A screenshot of a computer

Description automatically generated

### %core\_list\_dataset\_metadata

Purpose: list metadata of given datasets.

filename meta "&project\_folder/json/core\_dataset\_metadata.json";

%***core\_list\_dataset\_metadata***(

dataset\_path = %str

(&project\_folder/testdata/sdtm/dm.xpt,

&project\_folder/testdata/sdtm/ae.xpt,

&project\_folder/testdata/sdtm/ex.xpt,

&project\_folder/testdata/sdtm/lb.xpt),

output = %sysfunc(pathname(meta))

);

**data** \_null\_;

rc = jsonpp('meta','log');

**run**;

libname jsonfile json fileref=meta ordinalcount=none;

**data** metadata.core\_dataset\_metadata;

set jsonfile.root;

**run**;

filename meta clear;

libname jsonfile clear;

The macro extracts a JSON file with the dataset metadata. The JSON file can be easily converted to a SAS dataset as the code demonstrates.

Figure 11 core\_dataset\_metadata with dataset metadata

A screenshot of a computer

Description automatically generated

### %core\_list\_rule\_sets

Purpose: list the rule sets available in the cache.

filename rulesets "&project\_folder/json/core\_rule\_sets.json";

%***core\_list\_rule\_sets***(

output = %sysfunc(pathname(rulesets)),

cache\_path = &project\_folder/resources/cache

);

**data** \_null\_;

rc = jsonpp('rulesets','log');

**run**;

libname jsonfile json fileref=rulesets;

**data** metadata.core\_rulesets(keep=standard version);

length standard $**32** version $**16**;

set jsonfile.alldata;

standard = strip(scan(value, **1**, ','));

version = strip(scan(value, **2**, ','));

**run**;

**proc** **sort** data = metadata.core\_rulesets;

by standard version;

**run**;

filename rulesets clear;

libname jsonfile clear;

The macro extracts a JSON file with the available CORE rulesets in the cache. The JSON file can be easily converted to a SAS dataset as the code demonstrates. The dataset has two variables (standard and version) with the CORE rulesets.

Figure 12 core\_rulesets dataset with available CORE rulesets

A screenshot of a computer

Description automatically generated

### %core\_list\_rules

Purpose: list the rules available in the cache.

filename rules "&json\_folder/core\_rules\_&core\_standard.-&core\_standard\_version..json";

%***core\_list\_rules***(

output = %sysfunc(pathname(rules)),

standard = &core\_standard,

version = &core\_standard\_version,

cache\_path = &project\_folder/resources/cache

);

The **core\_list\_rules** macro extracts a JSON file with the available CORE rules for a given standard and version in the cache. By utilizing the dataset with CORE rulesets, we can extract all rules, organized by standard. We also utilize a macro (**get\_core\_rules**) to turn the JSON file into a SAS dataset (Figure 13).

**data** \_null\_;

set metadata.core\_rulesets;

length code $ **1024**;

if upcase(standard) = "DDF" then

%\* For DDF only get JSON ;

code = cats('%nrstr(%get\_core\_rules(core\_standard=', lowcase(standard), ', core\_standard\_version=', version, ', dsout=));');

else

code = cats('%nrstr(%get\_core\_rules(core\_standard=', lowcase(standard), ', core\_standard\_version=', version, '));');

put code=;

call execute(code);

**run**;

This dataset can be used to select rules when validating data, based on standard, standard version, class, domain, and dataset.

Figure 13 core\_rules dataset with available CORE rules

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

### %core\_validate\_data

Purpose: validate data.

%***core\_validate\_data***(

cache\_path = &project\_folder/resources/cache,

data= &project\_folder/testdata/sdtm,

standard = sdtmig,

version = **3**-**3**,

controlled\_terminology\_package = %str(sdtmct-**2023**-**12**-**15**),

output= &project\_folder/reports/&report\_name.\_sdtmig\_3-**3**,

output\_format = %str(XLSX, JSON),

raw\_report = **0**,

data\_format = XPT,

define\_xml\_path = &project\_folder/testdata/sdtm/define.xml,

whodrug = &project\_folder/testdata/dictionaries/whodrug,

meddra = &project\_folder/testdata/dictionaries/meddra,

rules =

);

In this example we validate all XPT files in the &project\_folder/testdata/sdtm folder,and we are not limiting the rules (rules =). The result will be an Excel spreadsheet similar to the one in Figure 8 together with a JSON file that contains the same content as in the Excel spreadsheet.

Using the core\_rules dataset from Figure 12, we can limit the validation to rules that are specific to datasets we want to validate. An example is below. A macro variable **core\_rules** is created that contains the applicable rules. We use this macro variable in the **%core\_validate\_data** macro parameter:  
 rules = "&core\_rules"

/\* Example of selecting rules \*/

**proc** **sql** noprint;

select distinct core\_id into :core\_rules separated by ','

from metadata.core\_rules

where (domains\_include in ('ALL' 'AE' 'DM')) and   
 (domains\_exclude ne 'DM') and   
 (domains\_exclude ne 'AE')and  
 (core\_standard = "sdtmig" and core\_standard\_version = "3-3")

order by core\_id;

**quit**;

options noquotelenmax;

%***core\_validate\_data***(

cache\_path = &project\_folder/resources/cache,

dataset\_path = %str

(&project\_folder/testdata/sdtm/dm.xpt,

&project\_folder/testdata/sdtm/ae.xpt),

standard = sdtmig,

version = **3**-**3**,

controlled\_terminology\_package = %str(sdtmct-**2023**-**12**-**15**),

output= &project\_folder/reports/&report\_name.\_sdtmig\_3-**3**,

output\_format = %str(XLSX, JSON),

raw\_report = **0**,

data\_format = XPT,

define\_xml\_path = &project\_folder/testdata/sdtm/define.xml,

whodrug = &project\_folder/testdata/dictionaries/whodrug,

meddra = &project\_folder/testdata/dictionaries/meddra,

rules = "&core\_rules"

);

The result will be an Excel spreadsheet similar to the one in Figure 8 together with a JSON file that contains the same content as in the Excel spreadsheet.

# Conclusion

This paper shows that it is possible to implement CDISC CORE Engine CLI commands in the SAS environment. By creating PROC FCMP functions or subroutines that contain Python functions that implement CORE commands, SAS macros can call these Python functions from the DATA step. Named parameters and default values can be easily implemented in SAS macros.

# References

[1] CDISC CORE – Available at <https://www.cdisc.org/core>

[2] de Donder, Nick & Peter van Reusel. (2023). CDISC Conformance rules and the CDISC Open Rules Engine Continuing the Road to Adoption. Proceedings of PHUSE EU Connect 2023.  
Retrieved from <https://phuse.s3.eu-central-1.amazonaws.com/Archive/2023/Connect/EU/Birmingham/PAP_SI04.pdf>

[3] CDISC CORE Engine GitHub Repository. Available at   
<https://github.com/cdisc-org/cdisc-rules-engine>

[4] YAML: YAML Ain't Markup Language™.   
<https://yaml.org/>

[5] CDISC CORE Rules Editor GitHub Repository.   
Available at <https://github.com/cdisc-org/conformance-rules-editor>

[6] Wyckmans, Marisa & Els Janssen (2023). Developing and Implementing CDISC CORE rules: from zero to hero. Proceedings of PHUSE EU Connect 2023.  
Retrieved from: <https://phuse.s3.eu-central-1.amazonaws.com/Archive/2023/Connect/EU/Birmingham/PAP_SI05.pdf>

[7] CDISC CORE Engine GitHub Repository Releases.  
Available at <https://github.com/cdisc-org/cdisc-rules-engine/releases>

[8] cdisc-rules-engine on the Python Package Index (PyPI).   
Available at <https://pypi.org/project/cdisc-rules-engine/>

[9] pickle — Python object serialization. Retrieved from <https://docs.python.org/3/library/pickle.html>

[10] Getting Started: Access to CDISC Library API using API Key Authentication.  
<https://wiki.cdisc.org/display/LIBSUPRT/Getting+Started%3A+Access+to+CDISC+Library+API+using+API+Key+Authentication>

[11] Zizzi, Mike (2019). SAS or Python? Why not use both? Using Python functions inside SAS programs. Retrieved from:   
<https://blogs.sas.com/content/sgf/2019/06/04/using-python-functions-inside-sas-programs/>

[12] SAS Institute. "FCMP Procedure." *Base SAS Procedures Guide*. Available at <https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/proc/n0pio2crltpr35n1ny010zrfbvc9.htm>

[13] SAS Institute. "Configuring SAS to Run the Python Language." Configuring SAS to Run External Languages. Available at <https://go.documentation.sas.com/doc/en/bicdc/9.4/biasag/n1mquxnfmfu83en1if8icqmx8cdf.htm>

[14] SAS Institute. "Using PROC FCMP Python Objects." *Base SAS Procedures Guide*. Available at  
<https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lecompobjref/p18qp136f91aaqn1h54v3b6pkant.htm>

[15] SAS Institute. "Python Object Language Elements." *Base SAS Procedures Guide*. Available at  
<https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/lecompobjref/p0x7optrcqo9vmn119q1yhxpj35n.htm>

[16] “click” on the Python Package Index (PyPI). Available at <https://pypi.org/project/click/>

# Contact Information

Your comments and questions are valued and encouraged. Contact the author at:

Lex Jansen

Sr Director, Data Science Development, CDISC (contract through Lex Jansen Consulting LLC)

Email: [ljansen@cdisc.org](mailto:ljansen@cdisc.org) or [lexjansen@gmail.com](mailto:lexjansen@gmail.com)

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.

# Appendices

## Appendix 1 : Code in the core-rules-engine Python file core.py to implement the validate command

import asyncio

import json

import logging

import os

import pickle

from datetime import datetime

from multiprocessing import freeze\_support

from typing import Iterable, Tuple

import click

from pathlib import Path

from cdisc\_rules\_engine.config import config

from cdisc\_rules\_engine.constants.define\_xml\_constants import DEFINE\_XML\_FILE\_NAME

from cdisc\_rules\_engine.enums.default\_file\_paths import DefaultFilePaths

from cdisc\_rules\_engine.enums.progress\_parameter\_options import ProgressParameterOptions

from cdisc\_rules\_engine.enums.report\_types import ReportTypes

from cdisc\_rules\_engine.models.validation\_args import Validation\_args

from cdisc\_rules\_engine.models.test\_args import TestArgs

from scripts.run\_validation import run\_validation

from scripts.test\_rule import test as test\_rule

from cdisc\_rules\_engine.services.cache.cache\_populator\_service import CachePopulator

from cdisc\_rules\_engine.services.cache.cache\_service\_factory import CacheServiceFactory

from cdisc\_rules\_engine.services.cdisc\_library\_service import CDISCLibraryService

from cdisc\_rules\_engine.utilities.utils import (

    generate\_report\_filename,

    get\_rules\_cache\_key,

)

from scripts.list\_dataset\_metadata\_handler import list\_dataset\_metadata\_handler

from version import \_\_version\_\_

def valid\_data\_file(file\_name: str, data\_format: str):

    fn = os.path.basename(file\_name)

    return fn.lower() != DEFINE\_XML\_FILE\_NAME and fn.lower().endswith(

        f".{data\_format.lower()}"

    )

@click.group()

def cli():

    pass

@click.command()

@click.option(

    "-ca",

    "--cache",

    default=DefaultFilePaths.CACHE.value,

    help="Relative path to cache files containing pre loaded metadata and rules",

)

@click.option(

    "-ps",

    "--pool-size",

    default=10,

    type=int,

    help="Number of parallel processes for validation",

)

@click.option(

    "-d",

    "--data",

    required=False,

    help="Path to directory containing data files",

)

@click.option(

    "-dp",

    "--dataset-path",

    required=False,

    multiple=True,

    help="Absolute path to dataset file",

)

@click.option(

    "-l",

    "--log-level",

    default="disabled",

    type=click.Choice(["info", "debug", "error", "critical", "disabled", "warn"]),

    help="Sets log level for engine logs, logs are disabled by default",

)

@click.option(

    "-rt",

    "--report-template",

    default=DefaultFilePaths.EXCEL\_TEMPLATE\_FILE.value,

    help="File path of report template to use for excel output",

)

@click.option(

    "-s", "--standard", required=True, help="CDISC standard to validate against"

)

@click.option(

    "-v", "--version", required=True, help="Standard version to validate against"

)

@click.option(

    "-ct",

    "--controlled-terminology-package",

    multiple=True,

    help=(

        "Controlled terminology package to validate against, "

        "can provide more than one"

    ),

)

@click.option(

    "-o",

    "--output",

    default=generate\_report\_filename(datetime.now().isoformat()),

    help="Report output file destination",

)

@click.option(

    "-of",

    "--output-format",

    multiple=True,

    default=[ReportTypes.XLSX.value],

    type=click.Choice(ReportTypes.values(), case\_sensitive=False),

    help="Output file format",

)

@click.option(

    "-rr",

    "--raw-report",

    default=False,

    show\_default=True,

    is\_flag=True,

    help="Report in a raw format as it is generated by the engine. "

    "This flag must be used only with --output-format JSON.",

)

@click.option(

    "-dv",

    "--define-version",

    help="Define-XML version used for validation",

)

@click.option(

    "-df",

    "--data-format",

    help="Format in which data files are presented. Defaults to XPT.",

    type=click.Choice(["xpt"], case\_sensitive=False),

    default="xpt",

    required=True,

)

@click.option("--whodrug", help="Path to directory with WHODrug dictionary files")

@click.option("--meddra", help="Path to directory with MedDRA dictionary files")

@click.option("--rules", "-r", multiple=True)

@click.option(

    "-p",

    "--progress",

    default=ProgressParameterOptions.BAR.value,

    type=click.Choice(ProgressParameterOptions.values()),

    help=(

        "Defines how to display the validation progress. "

        'By default a progress bar like "[████████████████████████████--------]   78%"'

        "is printed."

    ),

)

@click.option("-dxp", "--define-xml-path", required=False, help="Path to Define-XML")

@click.pass\_context

def validate(

    ctx,

    cache: str,

    pool\_size: int,

    data: str,

    dataset\_path: Tuple[str],

    log\_level: str,

    report\_template: str,

    standard: str,

    version: str,

    controlled\_terminology\_package: Tuple[str],

    output: str,

    output\_format: Tuple[str],

    raw\_report: bool,

    define\_version: str,

    data\_format: str,

    whodrug: str,

    meddra: str,

    rules: Tuple[str],

    progress: str,

    define\_xml\_path: str,

):

    """

    Validate data using CDISC Rules Engine

    Example:

    python core.py -s SDTM -v 3.4 -d /path/to/datasets

    """

    # Validate conditional options

    logger = logging.getLogger("validator")

    if raw\_report is True:

        if not (len(output\_format) == 1 and output\_format[0] == ReportTypes.JSON.value):

            logger.error(

                "Flag --raw-report can be used only when --output-format is JSON"

            )

            ctx.exit()

    cache\_path: str = os.path.join(os.path.dirname(\_\_file\_\_), cache)

    if data:

        if dataset\_path:

            logger.error(

                "Argument --dataset-path cannot be used together with argument --data"

            )

            ctx.exit()

        dataset\_paths: Iterable[str] = [

            str(Path(data).joinpath(fn))

            for fn in os.listdir(data)

            if valid\_data\_file(fn, data\_format)

        ]

    elif dataset\_path:

        if data:

            logger.error(

                "Argument --dataset-path cannot be used together with argument --data"

            )

            ctx.exit()

        dataset\_paths: Iterable[str] = [

            dp for dp in dataset\_path if valid\_data\_file(dp, data\_format)

        ]

    else:

        logger.error(

            "You must pass one of the following arguments: --dataset-path, --data"

        )

        # no need to define dataset\_paths here, the program execution will stop

        ctx.exit()

    run\_validation(

        Validation\_args(

            cache\_path,

            pool\_size,

            dataset\_paths,

            log\_level,

            report\_template,

            standard,

            version,

            set(controlled\_terminology\_package),  # avoiding duplicates

            output,

            set(output\_format),  # avoiding duplicates

            raw\_report,

            define\_version,

            data\_format.lower(),

            whodrug,

            meddra,

            rules,

            progress,

            define\_xml\_path,

        )

    )

## Appendix 2 : Python function called by SAS PROC FCMP to implement the validate command

def core\_validate\_data(cache, pool\_size, data, dataset\_path, log\_level, report\_template, standard, version, output, output\_format, raw\_report, controlled\_terminology\_package, define\_version, data\_format, define\_xml\_path, whodrug, meddra, rules):

      """Output: message\_return\_value"""

      import os

      import sys

      # Add top-level folder to path so that project folder can be found

      core\_path = os.environ["CORE\_PATH"]

      lib\_path = os.path.abspath(os.path.join(\_\_file\_\_, core\_path))

      if lib\_path not in sys.path: sys.path.append(lib\_path)

      import asyncio

      import json

      import logging

      import os

      import pickle

      from datetime import datetime

      from multiprocessing import freeze\_support

      from typing import Tuple

      import re

      from pathlib import Path

      from cdisc\_rules\_engine.config import config

      from cdisc\_rules\_engine.constants.define\_xml\_constants import DEFINE\_XML\_FILE\_NAME

      from cdisc\_rules\_engine.enums.default\_file\_paths import DefaultFilePaths

      from cdisc\_rules\_engine.enums.progress\_parameter\_options import ProgressParameterOptions

      from cdisc\_rules\_engine.enums.report\_types import ReportTypes

      from cdisc\_rules\_engine.models.validation\_args import Validation\_args

      from scripts.run\_validation import run\_validation

      from cdisc\_rules\_engine.services.cache.cache\_populator\_service import CachePopulator

      from cdisc\_rules\_engine.services.cache.cache\_service\_factory import CacheServiceFactory

      from cdisc\_rules\_engine.services.cdisc\_library\_service import CDISCLibraryService

      from cdisc\_rules\_engine.utilities.utils import (

          generate\_report\_filename,

          get\_rules\_cache\_key,

      )

      from scripts.list\_dataset\_metadata\_handler import list\_dataset\_metadata\_handler

      from version import \_\_version\_\_

      def valid\_data\_file(file\_name: str, data\_format: str):

          fn = os.path.basename(file\_name)

          return fn.lower() != DEFINE\_XML\_FILE\_NAME and fn.lower().endswith(

              f".{data\_format.lower()}"

          )

      def validate(

          standard: str,

          version: str,

          cache: str = core\_path + "/" + DefaultFilePaths.CACHE.value,

          pool\_size: int =10,

          log\_level: str = 'disabled',

          data: str = '',

          dataset\_path: Tuple[str] =[],

          report\_template: str = core\_path + "/" + DefaultFilePaths.EXCEL\_TEMPLATE\_FILE.value,

          output\_format: Tuple[str] = [ReportTypes.XLSX.value],

          raw\_report: bool = True,

          output: str = generate\_report\_filename(datetime.now().isoformat()),

          controlled\_terminology\_package: Tuple[str] = [],

          define\_version: str = '',

          data\_format: str = "XPT",

          rules: Tuple[str] = [],

          define\_xml\_path: str = '',

          whodrug: str ='',

          meddra: str = '',

          progress: str = 'disabled'

      ):

          """

          Validate data using CDISC Rules Engine

          Example:

          python core.py -s SDTM -v 3.4 -d /path/to/datasets

          """

          validation\_message = ""

          dataset\_path = [item.strip(' ') for item in dataset\_path if item !='']

          output\_format = [item.strip(' ') for item in output\_format if item !='']

          controlled\_terminology\_package = [item.strip(' ') for item in controlled\_terminology\_package if item !='']

          rules = [item.strip(' ') for item in rules if item !='']

          if not log\_level:

              log\_level = 'disabled'

          # Validate conditional options

          logger = logging.getLogger("validator")

          if raw\_report is True:

              if not (len(output\_format) == 1 and output\_format[0] == ReportTypes.JSON.value):

                  logger.error(

                      "Flag --raw-report can be used only when --output-format is JSON"

                  )

                  validation\_message = "Flag --raw-report can be used only when --output-format is JSON"

                  return validation\_message

          cache\_path: str = os.path.join(os.path.dirname(\_\_file\_\_), cache)

          if data:

              if dataset\_path:

                  logger.error(

                      "Argument --dataset-path cannot be used together with argument --data"

                  )

                  validation\_message = "Argument --dataset-path cannot be used together with argument --data"

                  return validation\_message

              dataset\_paths: Iterable[str] = [

                  str(Path(data).joinpath(fn))

                  for fn in os.listdir(data)

                  if valid\_data\_file(fn, data\_format)

              ]

          elif dataset\_path:

              if data:

                  logger.error(

                      "Argument --dataset-path cannot be used together with argument --data"

                  )

                  validation\_message = "Argument --dataset-path cannot be used together with argument --data"

                  return validation\_message

              dataset\_paths: Iterable[str] = [

                  dp for dp in dataset\_path if valid\_data\_file(dp, data\_format)

              ]

          else:

              logger.error(

                  "You must pass one of the following arguments: --dataset-path, --data"

              )

              validation\_message = "You must pass one of the following arguments: --dataset-path, --data"

              # no need to define dataset\_paths here, the program execution will stop

              return validation\_message

          run\_validation(

              Validation\_args(

                  cache\_path,

                  pool\_size,

                  dataset\_paths,

                  log\_level,

                  report\_template,

                  standard,

                  version,

                  set(controlled\_terminology\_package),  # avoiding duplicates

                  output,

                  set(output\_format),  # avoiding duplicates

                  raw\_report,

                  define\_version,

                  data\_format.lower(),

                  whodrug,

                  meddra,

                  rules,

                  progress,

                  define\_xml\_path,

              )

          )

          return validation\_message

      return\_message = validate(

           cache=cache,

           pool\_size=int(pool\_size),

           data=data,

           dataset\_path=re.split(';|,', dataset\_path),

           log\_level=log\_level,

           report\_template=report\_template,

           standard=standard,

           version=version,

           output=output,

           output\_format=re.split(';|,', output\_format),

           raw\_report=(raw\_report == 1),

           controlled\_terminology\_package=re.split(';|,', controlled\_terminology\_package),

           define\_version=define\_version,

           data\_format=data\_format,

           whodrug=whodrug,

           meddra=meddra,

           rules=re.split(';|,', rules)

       )

      return return\_message

1. The repository related to this paper (<https://github.com/lexjansen/cdisc-core-sas>) includes the latest release of the cdisc-rules-engine repository (<https://github.com/cdisc-org/cdisc-rules-engine>) in the cdisc-rules-engine folder. The code related to this paper has been tested with the CORE\_PATH environment variable pointing to this folder. You can point the CORE\_PATH environment variable to a development version of the cdisc-rules-engine repository. However, you may have to update Python functions and macros to make the code related to this paper work with this development version. It is the intention of the author to update the code in <https://github.com/lexjansen/cdisc-core-sas> when new release of the CORE engine are released. [↑](#footnote-ref-1)