

Chasing Master Data Interoperability: Facilitating Master Data Management (MDM) Objectives Through CSV Control Tables that Contain Data Rules that Support SAS® and Python Data-Driven Software Design

Troy Martin Hughes

ABSTRACT

Control tables are tabular data structures that contain *control data*—the data that direct software execution and which can prescribe dynamic software functionality. Control tables offer a preferred alternative to hardcoded conditional logic statements, which require code customization to modify. Thus, control tables can dramatically improve software maintainability and configurability by empowering developers and, in many cases, nontechnical end users to alter software functionality without having to modify code. Moreover, when control tables are maintained within canonical file structures such as comma-separated values (CSV), extensible markup language (XML), or JavaScript object notation (JSON), these ubiquitous data structures facilitate master data interoperability by enabling one control table to drive not only SAS software but also non-SAS applications (such as Python). This text introduces a reusable method that preloads CSV control tables into SAS temporary arrays to facilitate the evaluation of business rules and other data rules within SAS data sets. To demonstrate the interoperability of canonical data structures, including CSV control tables, a functionally equivalent Python program also ingests these control tables. Master data management (MDM) objectives are facilitated because only one instance of the master data—the control table, aka the single source of truth—is maintained, yet it can drive limitless processes across varied applications and programming languages. And, when data rules inevitably must be updated, the control data within the control table can be changed only once to effect corresponding changes in all derivative uses of those master data.

DATA-DRIVEN SOFTWARE DESIGN

Control tables provide a data-driven method to direct and modify software functionality by maintaining software instructions inside of tables rather than as code. This alternative to hardcoded software design can be further improved by maintaining control tables in interoperable (aka, *canonical*) file formats such as Excel spreadsheets, CSV files, XML files, or JSON files that can be accessed by not only SAS but also other applications and programming languages. Thus, the initial SAS solution that is demonstrated is immediately followed by a functionally equivalent Python solution—with both methods producing identical results while relying on the same underlying control data to drive all processing.

For this text, consider the requirement to scan SAS program files for specific SAS language elements such as SAS procedure names or SAS macro statements. In this scenario, a supervisor, product owner, or other key stakeholder might need to understand the breadth of language elements being used across a team of developers. And this real-life exercise occurred, when I was commissioned to analyze millions of lines of SAS code for an organization and to determine what, if any, SAS add-on products (such as SAS/OR and SAS/IML) could be eliminated from the budget. Thus, I created a mapping of every add-on product to its respective specialized procedures, statements, functions, and subroutines, and delivered an analysis demonstrating which add-on products were being used, to what extent they were being used, and ultimately what products could be removed from the exorbitant SAS license. That's right, using SAS to axe SAS!

The analysis could have been hardcoded, with each of the SAS add-on products and their respective procedures, statements, and functions typed into a SAS program file. However, a far more elegant, configurable, and maintainable solution instead recognizes that this taxonomy of syntax should instead be placed inside a control table that is interpreted by software. That is, in data-driven programming, these taxonomies, business rules, and other logic elements are *interpreted by* the software rather than *contained within* the software. And, most importantly, this added level of abstraction ensures that as business rules, data rules, taxonomies, or other logic elements are required to be updated, the code can remain secure and stable without the need to modify it—only the external data structures require change.

ARRAY-BASED INGESTION OF CONTROL TABLES TO EVALUATE TEXT FILES

To build a syntax interpreter in SAS, one data-driven solution is to maintain a control table of all syntax terms, to load that control table into memory, and to interrogate one or more SAS program files for each of the terms. Temporary arrays are useful because the entire control table can be loaded at the start of a DATA step, after which those arrays operate as the taxonomy used to parse one or thousands of separate SAS program files.

A CSV control table (e.g., d:\sas\terms.csv) contains an abbreviated list of sample terms:

```
PROC SORT,Procedure: Base
PROC FREQ,Procedure: Stats
PROC MEANS,Procedure: Stats
%MACRO,Macro Statement
%MEND,Macro Statement
%DO,Macro Statement
%LET,Macro Statement
```

Each row of the control table includes two comma-delimited values—the term itself and its user-specified classification. Search terms can be added, removed, or modified by altering only the control table, so this data-driven design facilitates software stability and integrity—because the underlying code can remain static while its control data are modified over time. Moreover, this data-driven design supports software configurability because different users can utilize the table to search for different terms; one user might want to scour SAS programs for built-in procedures, another for built-in functions, and a third for SAS macro statements. Each of these users can utilize the identical SAS program to perform these disparate operations—by modifying *only* control data. And that is the beauty of *configuration* over *customization*, where the former serves the needs of many while latter serves the needs of just one.

A sample SAS program file (d:\sas\etl.sas) will not be run but can be interrogated for these syntax keyword terms:

```
data etl;
  set pres;
  length fullname $50 term 8 numVPs 8;
  format term 8.2;
  fullname=catx(' ',fname,lname);
  term=round((dt2-dt1)/365.25,.01);
  numVPs=countw(vp,',');
run;

proc sort data=etl;
  by num;
run;

proc means data=etl;
  var term numVPs;
run;

proc freq data=etl;
  tables term numVPs;
run;

PROC FREQ data=etl;
  tables lname;
run;

%macro sample();
%put Today is %sysfunc(putn(%sysfunc(date()),mmddyy10.));
%mend;

%sample;
```

The CTRL_TEXT_DOMAIN_TEXT macro first ingests the control table to determine the number of observations; this step is required because the observation count is used subsequently in the declaration of temporary SAS arrays. A second DATA step first ingests the control table (when _N_=1) and subsequently ingests the program file to search for the terms:

```
* saved as d:\sas\ctrl_text_domain_text.sas;
%macro ctrl_text_domain_text(ctrl=/* ctrl tab CSV */,
    domain=/* SAS program file name to parse */,
    case=/* SENSITIVE to do case-sensitive search */);
* facilitate case-insensitive FIND;
%if "%upcase(&case)"="SENSITIVE" %then %let case=;
%else %let case='i';
* get observation count;
%local nobs;
data _null_;
    infile "&ctrl" trunccover end=eof;
    input line $500;
    if eof then call symputx('nobs',strip(put(_n_,8.)),'l');
run;

data metrics (drop=i term);
    if _n_=1 then do;
        i=0;
        infile "&ctrl" trunccover end=eof dsd delimiter=',';
        do until(eof);
            i=i+1;
            length term $32 cat $32;
            input term $ cat $;
            array arrkey[&nobs] $32 _temporary_;
            arrkey[i]=strip(term);
            array arrcat[&nobs] $32 _temporary_;
            arrcat[i]=strip(cat);
            end;
        end;

        length line 8 contents $500 keyword $32;
        infile "&domain" trunccover end=eof;
        input contents $500.;
        do i=1 to dim(arrkey);
            if find(contents,strip(arrkey[i])&case) then do;
                line=_n_;
                keyword=arrkey[i];
                cat=arrcat[i];
                output;
            end;
        end;
    end;
run;
%mend;
```

Within the _N_=1 DO block, the control table is ingested line by line, with each row mapped to a new array element. Because the control table comprises pairs of comma-delimited values, two temporary arrays are required, both of which will be declared (using the ARRAY statement) as having the identical number of elements (&NOBS). That is, once these twin arrays are declared and initialized, they can be iterated in parallel, with each successive array element representing a successive row in the control table. And because temporary arrays are not saved to the resultant data set (i.e., Metrics), this efficient solution minimizes the both CPU and I/O resources.

In the second DO loop, the twin arrays are iterated, with the SAS keyword (e.g., PROC SORT) mapped from the ARRKEY array and the keyword category (e.g., Procedure: Base) mapped from the ARRCAT array. The OUTPUT statement writes each occurrence of syntax keywords (discovered in the program file being analyzed) to the Metrics data set, with the program line number also recorded to ensure the syntax is readily discoverable.

Finally, and as is typical of data-driven software design, the preceding reusable macro is called by a driver program (d:\sas\driver.sas)—note that the user-specified file location (&LOC) must be modified, after which the macro can be executed:

```
%let loc=D:\sas\;          * USER MUST CHANGE LOCATION *;
%include "&loc.ctrl_text_domain_text.sas";
%ctrl_text_domain_text(ctrl=&loc.terms.csv,
    domain=&loc.etl.sas, case=);
```

Table 1 demonstrates the Metrics data set that is created from the default case-insensitive search.

Cat	Line	Contents	Keyword
Procedure: Base	10	proc sort data=etl;	PROC SORT
Procedure: Stats	14	proc means data=etl;	PROC MEANS
Procedure: Stats	18	proc freq data=etl;	PROC FREQ
Procedure: Stats	22	PROC FREQ data=etl;	PROC FREQ
Macro Statement	26	%macro sample();	%MACRO
Macro Statement	28	%mend;	%MEND

Table 1. Metrics Data Set with Search Results

The use of SAS arrays is instrumental to the macro because the arrays enable reserved words and special characters to be parsed without the need to mask them. For example, both %MACRO and %MEND are identified within the etl.sas program file without issue. Too often, the contents of control tables are unnecessarily exported from SAS data sets into macro variables or macro “lists.” Although this methodology may be preferred in some instances, character masking is often required and can complicate the solution.

Note that the CTRL_TEXT_DOMAIN_TEXT macro references only the structure—not the contents—of the underlying control table. Thus, the control table must be comma-delimited and contain two columns of data (because these columns are populated into the ARRKEY and ARRCAT arrays, respectively); however, the contents of these two columns is flexible and can be modified to meet the needs of specific users (i.e., configurability), or to adapt over time to changes in the SAS programming language itself (i.e., adaptability and future-proofing).

This data-driven methodology also facilitates scalability, in that as the list of keywords grows, the macro that reads these control data remains stable. Had a functionally equivalent hardcoded solution been utilized—for example, a gaggle of IF-THEN-ELSE statements to assess whether a keyword had been discovered—additional lines of code would need to have been added each time a new keyword needed to be located. By contrast, within a data-driven design paradigm, the increased software flexibility, adaptability, configurability, and scalability each contributes to increased *software maintainability*—the ease with which software can be maintained (and modified, as necessary).

The CTRL_TEXT_DOMAIN_TEXT macro is demonstrated and more fully described in the author's text: *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality, Second Edition*. (Hughes, 2022) The text additionally demonstrates other control table scenarios, such as maintaining a CSV control table and interrogating a SAS data set, and maintaining a SAS control table and interrogating either a text file or a SAS data set.

CONTROL TABLE INTEROPERABILITY – SUPPLANTING SAS WITH PYTHON

As mentioned, because the control table is maintained within a CSV file, it is more readily ingested by non-SAS applications such as Python. For example, another analyst might be given the same task (to find syntax terms within program files), and given Python's far more expansive and capable text-parsing functionality (as compared with Base SAS), choose to deliver a solution using Python. Note how much cleaner, simpler, and straightforward the Python code is, as compared to the preceding functionally equivalent SAS:

```

import csv
import os

loc = r'd:\sas'

# ingest control table into list
with open(os.path.join(loc, 'terms.csv')) as ctrl:
    reader=csv.reader(ctrl)
    terms=list(reader)

# initialize empty list
metrics=[]

# iterate the etl.sas program file to search for syntax keywords
with open(os.path.join(loc, 'etl.sas')) as f:
    datafile=f.readlines()
    for idx, line in enumerate(datafile, start=1):
        for term in terms:
            if term[0] in line.upper():
                metrics.append([idx, line, term[0]])

```

The Python program first imports the CSV control table into the **terms** list, after which the program file is parsed for syntax keywords and phrases. When a keyword is discovered, the phrase, the program line in which it was discovered, and the line number are appended to the **metrics** list. In a more comprehensive example, the name of the program file being interrogated would be abstracted, and a function wrapper created to an endless number of program files could be analyzed through the same concise code.

Note that master data management (MDM) objectives are met and principles are upheld through this data-driven design. Because the business rules—in this case, the control table containing the taxonomy of SAS syntax—were maintained outside of the code within a canonical data structure (i.e., a CSV file), the logic driving the functionality could be easily ported from SAS to Python. In fact, the Python solution only had to interpret and operationalize these rules to create a list containing the incidences of all SAS phrases and keywords that were discovered. That is, the control data did not need to be required to accomplish this portability. However, were changes to this taxonomy required, they could be made to the single control table and yet drive functionality in both the SAS and Python programs. MDM at its best!

CONCLUSION

The use of control tables epitomizes data-driven software design by storing software instruction within malleable data structures rather than statically within code. This text introduced an array-based method to ingest and evaluate control tables, overcoming a common weakness in which SAS control data are unnecessarily converted to macro variables and macro “lists.” Maintaining control tables in interoperable data structures such as CSV files further broadens the scope of the applications and programming languages that can leverage control tables. Furthermore, this promotes MDM objectives by maintaining a single version of the truth—the master control data driving software and providing dynamic functionality. This was patently demonstrated by the functionally equivalent (yet obviously superior) Python solution, which relied on the same underlying control data as the SAS solution.

REFERENCES

Hughes, T. M. (2022). *SAS® Data-Driven Development: From Abstract Design to Dynamic Functionality, Second Edition*. San Diego, California.

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com