Abstract
The EXECUTE call routine of SAS® can be used for many tasks that would otherwise need many lines of code. No more multiple steps to manage data to conform for macro reference, and no more repetitive, inefficient code: CALL EXECUTE allows the programmer to operate straight from the comfort of a single data step. We provide you with some examples of how CALL EXECUTE can transform a multiple-tiered SAS program to a compact, efficient, single-tier beauty!

Introduction
In the programming world, programmers often focus on writing compact, efficient, straightforward code. When faced with the task of performing one function multiple times, we SAS programmers turn to macro processing or successive conditional loops (if-then-else, do, do-while, ...). At times, we also manipulate data structure just so we can perform these macro invocations and successive conditional loops. Such data manipulation could involve outputting a string of text to be stored in a macro variable, or partitioning datasets for individual processing, or whatever else we need to do to conclude our program with efficient code, never mind the inefficient steps we took to get there. Many of these precursory data manipulations can be avoided by using the dataset containing the data that drive the repeated calls to a function, with one or more CALL EXECUTE statements. A function can be anything - a procedure, a macro, another data step process - and the function call can be conditional, based on the data values.

The CALL EXECUTE syntax
CALL EXECUTE is used within a data step, and unlike most macro conditionals, can be used in open code. It has the following syntax:

Call execute(argument);

where argument can be one of three elements:

1) a character string enclosed in quotation marks:

Call execute('Proc print data=work.demo; run;');

2) the name of a variable contained within the dataset processed within the data step:

Call execute('Proc print data=work.'||filename||'; run;');
3) a character expression that resolves to a SAS statement or macro text expression:

        Call execute('%printit(file='||left(filename)||'); run;');

**Procedure call**

A basic SAS® function is to print all data from a dataset.

    Proc print data=libref.aeq; Var _all_; Run;

This will provide a complete print of the dataset AEQ. This same operation can be done via CALL EXECUTE:

    Data _null_;  
        Dsname='aeq';  
        Call execute('Proc print data=libref.'||trim(left(dsname))||
            '; Var _all_; Run;');  
    Run;

You can see that the example is longer and more complicated than the initial print procedure. Why would any programmer want to replace three lines of code with four lines of code? This code enhancement is not an enhancement at all.

When printing multiple files, though, one CALL EXECUTE is more efficient than multiple prints:

    Proc print data=libref.AEQ; Var _all_; Run;  
    Proc print data=libref.CONMED; Var _all_; Run;  
    Proc print data=libref.DEMOG; Var _all_; Run;  
    Proc print data=libref.LAB; Var _all_; Run;  
    Proc print data=libref.TERM; Var _all_; Run;

Using CALL EXECUTE:

    Data _null_;  
        Do Dsname='aeq', 'conmed', 'demog', 'lab', 'term';  
            Call execute('Proc print data=libref.'||trim(left(dsname))||
                '; Var _all_; Run;');  
        end;  
    Run;

For reference, the same thing can be done using macro calls, but the code is nowhere near as compact:
%Macro printit(dataset=,dtvars=);
   Proc print data=libref.&dataset;
     Var _all_;
   Run;
%Mend printit;
%printit(dataset=aeq);
%printit(dataset=conmed);
%printit(dataset=demog);
%printit(dataset=lab);
%printit(dataset=term);

Consider a case in which we want a complete print of all the datasets in the library libref. You can do this automatically with CALL EXECUTE, using either PROC CONTENTS or PROC SQL to generate a file containing the names of the datasets in the library. This example uses SQL:

proc sql;
   create table ds as
     select distinct memname
     from dictionary.tables
     where libname="LIBREF" and memtype="DATA"
     order by memname;
quit;

data _null_;  
   set ds;
   call execute ('proc print data=libref.'||memname||"; run;");
run;

Now lets consider a case in which we want a print of the datasets within the library in which only date fields are to be printed, in format date9. We will print only those datasets that have at least one date field.

It would take a bit of coding to print each individual dataset’s date fields. Two quick options come to mind:

Use successive invocations of the print procedure:
Proc print data=libref.AEQ;
   Var STRTDT STOPDT date9.;
Proc print data=libref.CONMED;
   Var STRTDT STOPDT date9.;
Proc print data=libref.DEMOG;
   Var VISITDT date9.;
Proc print data=libref.LAB;
   Var SAMPDT LABDT date9.;
Proc print data=libref.TERM;
   Var LDRGDT CONDT TERMDT FUDT date9.;
Run;

Or if the programmer is comfortable with macro processing, build a macro:

%Macro printit(dataset=,dtvars=);
   Proc print data=libref.&dataset;
      Var &dtvars;
      Format &dtvars date9.;
   Run;
%Mend printit;

%printit(dataset=AEQ,dtvars= STRTDT STOPDT);
%printit(dataset=CONMED,dtvars= STRTDT STOPDT);
%printit(dataset=DEMOG,dtvars= VISITDT);
%printit(dataset=LAB,dtvars= SAMPDT LABDT);
%printit(dataset=TERM,dtvars= LDRGDT CONDT TERMDT FUDT);

Both of these routes require prior knowledge of the dataset names and date variables, knowledge acquired manually. Insertion of manual intervention leads to possible insufficiencies of code, such as if the database changes in structure in a way not anticipated by the program code.

Let’s build a dataset (DATEDDS.SD2) which contains the names of all datasets in the library libref, along with any date variables of the dataset. Assume date variables are those with names that end in text string 'DT'. A quick approach would be to process the SQL dictionary table COLUMNS and transpose the results to obtain one record per dataset with at least one stored date variable field.

Proc sql;
   Create table ds as
      Select distinct memname, name
      From dictionary.columns
      where libname='LIBREF' and
         substr(trim(name),length(trim(left(name)))-1,2)='DT'
      Order by memname, name;
Quit;
Proc transpose data=ds out=datedds prefix=DT;
   By memname;
   Var name;
Run;

The resulting DATEDDS dataset looks like this:

<table>
<thead>
<tr>
<th>Obs</th>
<th>MEMNAME</th>
<th>DT1</th>
<th>DT2</th>
<th>DT3</th>
<th>DT4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AEQ</td>
<td>STRTDT</td>
<td>STOPDT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>CONMED</td>
<td>STRTDT</td>
<td>STOPDT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>DEMOG</td>
<td>VISITDT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LAB</td>
<td>SAMPDT</td>
<td>LABDT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TERM</td>
<td>LDRGDT</td>
<td>CONDT</td>
<td>TERMDT</td>
<td>FUDT</td>
</tr>
</tbody>
</table>

Using DATEDDS, we can obtain the desired prints of all date variable fields to satisfy the original request entirely without manual intervention using CALL EXECUTE.

Data _null_;  
Set DATEDDS;  
call execute('Proc print data=libref.'||memname||'');  
call execute('  var '||dt1||' '||dt2||' '||dt3||' '||dt4||';');  
call execute('   format '||dt1||' '||dt2||' '||dt3||' '||dt4||' date9.;');  
call execute('Run;');  
Run;

Furthering this example, we can also conditionally invoke the print procedure based on select criteria. Let’s choose to print only those datasets’ date variable fields where the dataset contains the date variable field VISITDT.

Data _null_;  
Set DATEDDS;  
If dt1='VISITDT' or dt2='VISITDT' or dt3='VISITDT' or dt4='VISITDT' then do;  
call execute('Proc print data=libref.'||memname||'');  
call execute('  var '||dt1||' '||dt2||' '||dt3||' '||dt4||';');  
call execute('   format '||dt1||' '||dt2||' '||dt3||' '||dt4||' date9.;');  
call execute('Run;');  
End;  
Run;
Two notes of interest with this example:

- Successive CALL EXECUTE invocations can be used to complete SAS code. You can write an entire SAS program using successive CALL EXECUTE invocations! Note of caution, the argument of each CALL EXECUTE should not exceed 256 characters in that it may result in characters 257 and after being ignored. In addition, if the length of successive CALL EXECUTE invocations sums to 32,767 characters or greater, errors may also result.

- The resulting CALL EXECUTE statements of the above example will not execute until the data step completes. See the below SAS log:

```sas
16   options mprint mlogic;
17   Data _null_
18     Set DATEDDS;
19     call execute('Proc print data=libref.' || trim(left(memname)) || ';');
20     call execute(' var ' || trim(left(dt1)) || ' ' || trim(left(dt2)) ||!' ' || trim(left(dt3)) || ' ' || trim(left(dt4)) || ';');
21     call execute(' format ' || trim(left(dt1)) || ' ' || trim(left(dt2)) ||!' ' || trim(left(dt3)) || ' ' || trim(left(dt4)) || ' date9.;');
22     call execute('Run;');
23   Run;

NOTE: There were 6 observations read from the data set WORK.DATEDDS.
NOTE: DATA statement used:
real time           0.03 seconds
cpu time            0.03 seconds

NOTE: CALL EXECUTE generated line.
1     + Proc print data=libref.AEQ;
2     + var stopdt strtdt ;
3     + format stopdt strtdt date9.;
4     + Run;

NOTE: There were 376 observations read from the data set LIBREF.AEQ.
NOTE: The PROCEDURE PRINT printed pages 1-8.
NOTE: PROCEDURE PRINT used:
real time           0.06 seconds
cpu time            0.04 seconds

5     + Proc print data=libref.CONMED;
6     + var stopdt strtdt ;
7     + format stopdt strtdt date9.;
8     + Run;

NOTE: There were 239 observations read from the data set LIBREF.CONMED.
NOTE: PROCEDURE PRINT used:
real time           0.03 seconds
cpu time            0.00 seconds
```
Macro invocation

Using the example above with macro printit with a CALL EXECUTE, you could as well perform the following data step.

```
Data _null_;  
Set DATEDDS;  
call execute('%printit(dataset='||memname||
              ',dtvars='||dt1||' '||dt2||' '||dt3||' '||dt4||');');  
Run;
```

Since this call execute involves a macro invocation, the data step processing halts to invoke the macro statements, building the macro definitions, though the SAS statements of the defined macro are not executed until the data step fully completes. This can be seen via the SAS log, employing option mprint.
Data _null_;
set DATEDDS;
call execute('%printit(dataset='||trim(left(memname))||',dtvars='||
trim(left(dt1))||' ||trim(left(dt2))||'|' ||
trim(left(dt3))||' ||trim(left(dt4))||'|');
run;

MLOGIC(PRINTIT):  Beginning execution.
MLOGIC(PRINTIT):  Parameter DATASET has value AEQ
MLOGIC(PRINTIT):  Parameter DTVARS has value stopdt strtdt
MPRINT(PRINTIT):  Proc print data=libref.AEQ;
MPRINT(PRINTIT):  Var stopdt strtdt;
MPRINT(PRINTIT):  Format stopdt strtdt date9.;
MPRINT(PRINTIT):  Run;
MLOGIC(PRINTIT):  Ending execution.
MLOGIC(PRINTIT):  Beginning execution.
MLOGIC(PRINTIT):  Parameter DATASET has value CONMED
MLOGIC(PRINTIT):  Parameter DTVARS has value stopdt strtdt
MPRINT(PRINTIT):  Proc print data=libref.CONMED;
MPRINT(PRINTIT):  Var stopdt strtdt;
MPRINT(PRINTIT):  Format stopdt strtdt date9.;
MPRINT(PRINTIT):  Run;
MLOGIC(PRINTIT):  Ending execution.
MLOGIC(PRINTIT):  Beginning execution.
MLOGIC(PRINTIT):  Parameter DATASET has value DEMOG
MLOGIC(PRINTIT):  Parameter DTVARS has value VISITDT
MPRINT(PRINTIT):  Proc print data=libref.DEMOG;
MPRINT(PRINTIT):  Var VISITDT;
MPRINT(PRINTIT):  Format VISITDT date9.;
MPRINT(PRINTIT):  Run;
MLOGIC(PRINTIT):  Ending execution.
MLOGIC(PRINTIT):  Beginning execution.
MLOGIC(PRINTIT):  Parameter DATASET has value LAB
MLOGIC(PRINTIT):  Parameter DTVARS has value labdt sampdt
MPRINT(PRINTIT):  Proc print data=libref.LAB;
MPRINT(PRINTIT):  Var labdt sampdt;
MPRINT(PRINTIT):  Format labdt sampdt date9.;
MPRINT(PRINTIT):  Run;
MLOGIC(PRINTIT):  Ending execution.
NOTE: DATA statement used:
real time 0.01 seconds
cpu time 0.01 seconds

MLOGIC(PRINTIT):  Beginning execution.
MLOGIC(PRINTIT):  Parameter DATASET has value TERM
MLOGIC(PRINTIT):  Parameter DTVARS has value CONDT FUDT LDRGDT TERMDT
MPRINT(PRINTIT):  Proc print data=libref.TERM;
MPRINT(PRINTIT):  Var CONDT FUDT LDRGDT TERMDT;
MPRINT(PRINTIT):  Format CONDT FUDT LDRGDT TERMDT date9.;
MPRINT(PRINTIT):  Run;
MLOGIC(PRINTIT):  Ending execution.
NOTE: There were 6 observations read from the data set WORK.DATEDDS.
NOTE: CALL EXECUTE generated line.
1  + Proc print data=libref.AEQ; Var stopdt strtdt; Format stopdt strtdt date9.;
Run;

NOTE: There were 376 observations read from the data set LIBREF.AEQ.
NOTE: The PROCEEDURE PRINT printed pages 59-66.
NOTE: PROCEDURE PRINT used:
  real time           0.03 seconds
  cpu time            0.01 seconds

1  +
;
2  + Proc print data=libref.CONMED; Var stopdt strtdt; Format stopdt strtdt date9.; Run;

NOTE: There were 239 observations read from the data set LIBREF.CONMED.
NOTE: PROCEDURE PRINT used:
  real time           0.01 seconds
  cpu time            0.01 seconds

2  +
;
3  + Proc print data=libref.DEMOG; Var VISITDT; Format VISITDT date9.; Run;

NOTE: There were 53 observations read from the data set LIBREF.DEMOG.
NOTE: The PROCEDURE PRINT printed pages 72-73.
NOTE: PROCEDURE PRINT used:
  real time           0.00 seconds
  cpu time            0.00 seconds

3  +
;
4  + Proc print data=libref.LAB; Var labdt sampdt; Format labdt sampdt date9.; Run;

NOTE: There were 1935 observations read from the data set LIBREF.LAB.
NOTE: The PROCEDURE PRINT printed pages 78-115.
NOTE: PROCEDURE PRINT used:
  real time           0.06 seconds
  cpu time            0.01 seconds

4  +
;
5  + Proc print data=libref.TERM; Var CONDT FUDT LDRGDT TERMDT; Format CONDT
    FUDT LDRGDT
    TERMDT date9.; Run;

NOTE: There were 28 observations read from the data set LIBREF.TERM.
NOTE: The PROCEDURE PRINT printed page 116.
NOTE: PROCEDURE PRINT used:
  real time           0.01 seconds
  cpu time            0.01 seconds
This is different from execution of CALL EXECUTE arguments that are not macro based, as mentioned above, where no processing is done until the data step processing is complete.

**Other Limitations**

Macro variables defined in the CALL EXECUTE argument cannot be referenced until the data step processing the call is completed. For example,

```sas
%macro test_;
    data _null_;
    call symput('macrvar',100);
    run;
    %put Macro variable macrvar equals &macrvar;
%mend test_;

data _null_; call execute('%test_'); run;
```

The macro invocation, as discussed previously, will process the macro statements of `%test_` immediately. However, the call `symput` of the data step within `%test_` will not execute until the initial data step containing the CALL EXECUTE terminates; the macro variable `&macrvar` will not be assigned until its defining data step executes. With this, this execution of this code will return a warning in the log declaring the macro variable `&macrvar` as not resolved. Changing the code above to

```sas
%macro test_;
    data _null_;
    call symput('macrvar',100);
    run;
%mend test_;

data _null_; call execute('%test_'); run;
%put Macro variable MACRVAR equals &macrvar;
```

will display

```sas
Macro variable MACRVAR equals 100
```

Moving the `%put` statement outside of the macro, to be executed after the data step processing completes, allows `&macrvar` to be printed with no such warning.
Conclusion

CALL EXECUTE is a tremendously far-reaching functionality of SAS that allows us to build in open code successive calls to a SAS function, procedure, etc., via a single data step. It can eliminate excessive processing of data that is frequently needed for repetitive execution of a macro or procedure. We hope that you, our fellow SAS programmer, can take the examples above and try them from the comfort of your own desktop, applying CALL EXECUTE methodology and presumably eliminating a lot of programming code that you will not need when the power of CALL EXECUTE is substituted in!

References