PharmaSUG China 2019 - Paper AD-028

Programming Automation Using Object Oriented Python and Pandas

Dan Li, Sanofi-Aventis

ABSTRACT

Pharmaceutical companies have been striving to raise automation level of Biostatic programming.

A promising way is taking advantage of template programs.

However, two inconveniences will be introduced by using SAS version of template program: firstly, user of template program have to copy paste the program and delete unnecessary codes; secondly, if users want to adapt the program to peculiar study specific derivation logic, invasive code modification is needed which will cause potential trouble for further use.

This paper uses Python Object Oriented feature and Pandas (Python library for data manipulation) to refactor SAS ADaM template (ADSL) program. Reusability and extensibility of the refactored template program conquer the two flaws mentioned above. Integrated with metadata, programmers only need to modify the metadata file accordingly and write much less code. Automation level is subsequently raised.

INTRODUCTION

With years of practice, companies gradually build up their own SAS template programs to accelerate coding for ADaM datasets. In consideration of relatively routine part of clinical trial studies, template SAS programs are very handy to use. User of template SAS program only need to copy the template code and delete unnecessary logic branches according to the study requirements.

However, this process has two flaws. One is that the user need to copy and paste the template program to use it, rather than use the template directly in code. The other is that the template is not suited to accommodate unusual derivation logic. If derivation rule of some study is different than that in template program, user need to write the logic in the middle of somewhere in template program. This is inconvenient and makes code not reusable. For example, in a series of studies of a compound, for the same ADaM dataset, situations are similar but may differ a little for each study. If we copy and paste template program several times but changed the code here and there, it is prone to forget to change code somewhere or make mistakes when a modification is needed.

This paper presents using Object Oriented Programming ideology to overcome the two flaws mentioned above. *Pandas* is a famous Python library for data manipulation. It provides equivalents for all regular operations in base SAS. An example of ADSL is used to illustrate the ideology.

PANDAS INTRODUCTION

Pandas is a library of Python ecology, whose purpose is to manipulate, aggregate and provide easy to use analysis tools. More specifically, it provides data structures and operations for manipulating numerical tables and time series.

The core data structure for Pandas is called *DataFrame*. A typical DataFrame looks like below:

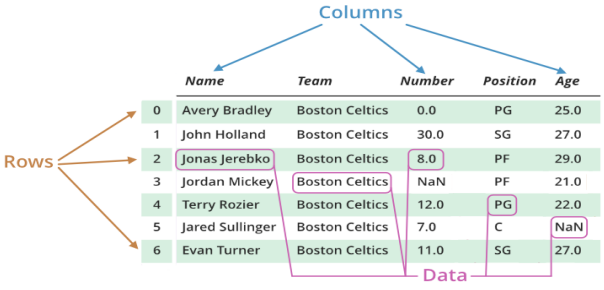
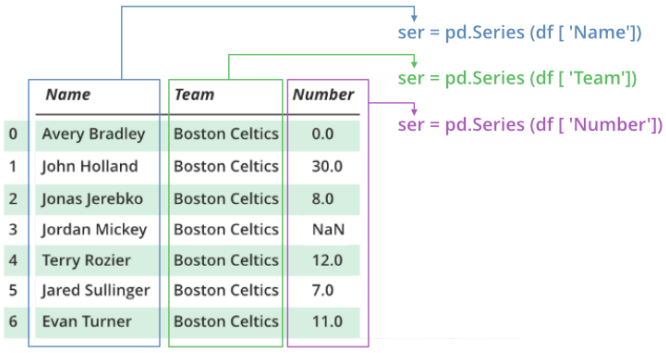


Figure 1. Pandas DataFrame

Typically, a *DataFrame* looks exactly like a SAS dataset: it has rows consist of different variables, different in types and values. While SAS datasets are processed by rows, Pandas *DataFrame* is composed of columns, or Pandas *Series*, a data structure looks like a column, but has an index with it (in Figure 2, index is 0, 1, 2, 3, 4, 5, 6; you could also define your own index).



	Name	Team	Number
0	Avery Bradley	Boston Celtics	0.0
1	John Holland	Boston Celtics	30.0
2	Jonas Jerebko	Boston Celtics	8.0
3	Jordan Mickey	Boston Celtics	NaN
4	Terry Rozier	Boston Celtics	12.0
5	Jared Sullinger	Boston Celtics	7.0
6	Evan Turner	Boston Celtics	11.0

Figure 2. Pandas Series

CLASS DESIGN AND OPERATION FLOW

Class design is how you use Object Oriented design to model your problem you want to solve. In this particular situation, we need to have an interaction among ADaM metadata, dataset and variables. Thus below class design:

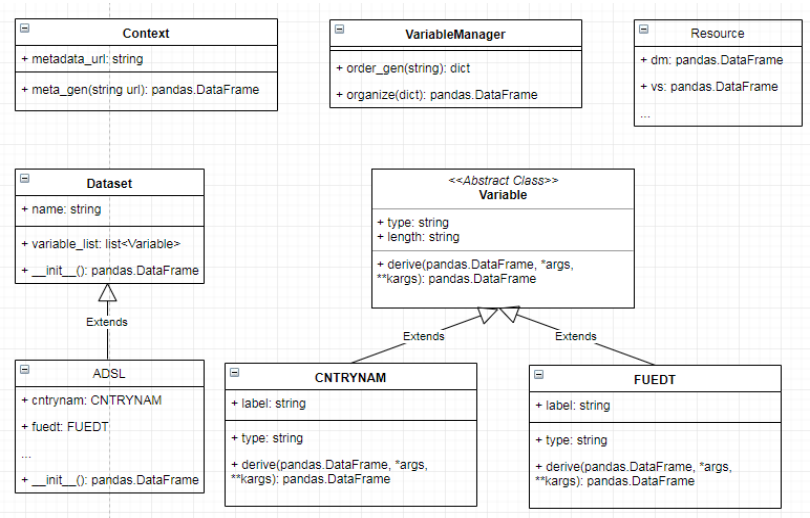


Figure 3. Class Design

Let's take ADSL as an example. Each variable in ADSL would be a class. For example for CNTRYNAM, there are two attributes: label and type, and has a method that contains the logic of how CNTRYNAM is derived. The function *derive()* will always take an input dataset (in the form of *DataFrame*) and other arbitrary arguments represented by *args and **kargs. And it will return a dataset in the form of *DataFrame*.

Class *Context* has an attribute metadata_url, which is the location of ADaM metadata Excel file. It has two functions: *meta_gen()* and *order_gen().* Function *meta_gen()* generates a *DataFrame* metadata.

Class *VariableManager* has two functions: *order_gen()* and *organize().* Function *order_gen()* will take the name of a dataset and search it from the metadata *DataFrame*, and generate the dependency relationship in the form of a tree-shaped data structure. The derivation order of variables could be obtained from iterating from bottom of the tree to the top of it. Function *organize()* will take in the dependency relationship and iterate through it. In the process of iterating, it calls each variable's *derive()*

function. Thus each variable's derivation logic is executed following an appropriate sequential order. In the end, the output *DataFrame* will have all variables derived.

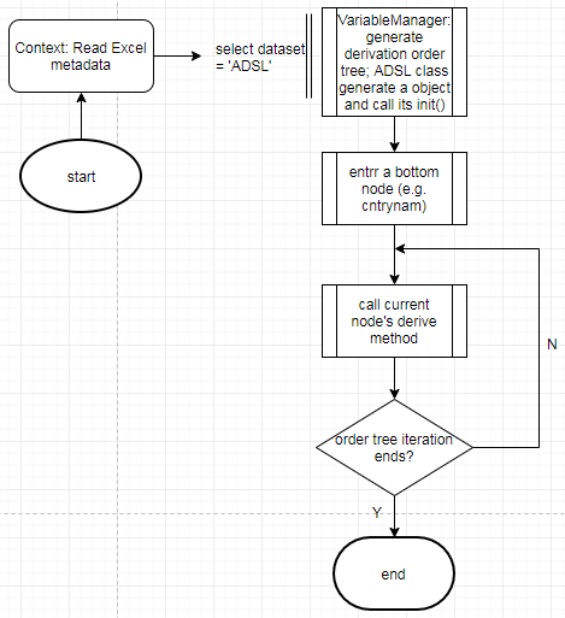


Figure 4. Operation Flow

KEY IMPLEMENTATION EXAMPLE

ADSL CLASS INITIALISATION

In Figure 4, before entering a bottom node of order tree and starting iteration, the function __init__() of class ADSL is called to merge to gather necessary variables from other datasets.

```
adsl = ADSL()
```

When creating the adsl object, class ADSL's *__init__()* function is automatically called:

```
def __init__(self):
    self.df = pd.merge(Resource.dm, Resource.vs, how='left', on='USUBJID')
```

Resource is a class with other domain datasets as its attributes. When *__init__()* function is called, *dm* dataset and *vs* dataset are merge by USUBJID and then the merged dataset is associated with attribute *df* of *adsl*. Other dataset merging may be needed according to study needs. Eventually, in function *__init__()* we get all 'ingredients' for further ADSL derivation.

CNTRYNAM AND FUEDT DERIVATION

CNTRYNAM (name of country) and FUEDT (end of follow-up date) are two variables in ADSL dataset. From SAS template we can have following equivalent function to implement derivation logic:

```
def cntrynam(self):
    df = pd.merge(Resource.dm, Resource.country_ref, how='left',
    left_on='COUNTRY', right_on='country')
    self.df = df.rename(columns={'cntrynam': 'CNTRYNAM'})
```

The first line does merging for dm and country_ref datasets. Then second line renames column name in merged dataset from 'cntrynam' to 'CNTRYNAM'.

While for variable FUEDT, in SAS template there are two possible situations, thus overloading is used here to dispatch control flow to the appropriate realization according to input parameters.

```
@overload
def fuedt(self, dm, ae):
    pass
@overload
def fuedt(self, dm):
    pass
def fuedt(self, dm, ae=None):
    if ae is not None:
          # convert AEENDTC to numeric value and assign to AEENDT
          ae_temp = ae[['STUDYID', 'USUBJID', 'AEENDTC']]
          ae_temp['AEENDT'] = ae_temp['AEENDTC'].astype('datetime64')
          # select distinct studyid, usubjid, max(aeendt) as _lstaeendt
          sub_table =
ae_temp.drop_duplicates().groupby([ae_temp['STUDYID'],
ae_temp['USUBJID']])
          sub_table = sub_table.rename(colunms={'AEENDT': '_lstaeendt'})
          self.df['RFPENDT'] = self.df['RFPENDTC'].astype('datetime64')
          self.df['FUEDT'] = self.df[['RFPENDT', '_lstaeendt']].max(axis=1)
    else:
          self.df['FUEDT'] = self.df['RFPENDTC'].astype(datetime64)
```

The 'overload' mechanism takes effect as: the first two functions having a '@overload' on top of itself acts like a 'protocol'. If the input parameter matches either one, it goes into corresponding implementation. The third *fuedt()* function is the implementation of both the first and the second protocol. If the input parameter does not have an *ae* parameter, then it matches the first protocol and goes into the first branch of if-else in the third function; if the input parameter has only a *dm* parameter, it goes into the second branch of if-else. In this way, the user does not need to distinguish details to select either derivation logic. The user always use the same function: *fuedt()*, just need to speficy input parameters.

If the user want to use the first derivation logic, he may just call function *fuedt()* as below:

```
adsl = ADSL()
adsl.fuedt(Resource.dm, Resource.ae)
```

Else if the user want to use the second derivation logic, he may just call *fuedt()* as below:

```
adsl.fuedt(Resource.dm)
```

If the user has his own derivation logic rather than the two implementations defined in class ADSL, he could implement his own in a sub-class of ADSL:

```
class ADSL_NEW(ADSL):
    def fuedt(self):
        a = 1
        b = 2
         print('This is the specific derivation logic')
adsl_new = ADSL_NEW()
adsl_new.fuedt()
```

In the new class ADSL_NEW inherited from ADSL, a *fuedt()* function is re-written with user-defined derivation logic. In this situation, the user does not need to invasively modify the template code. As template, class ADSL remains untouched. And user could feel free to add any study-specific derivation logic to derive variables.

ORDER GENERATION

After each variable's derivation function is ready, the order of variables could be obtained from metadata. From metadata, first parent and child lists as below is obtained:

```
   'parent': ['usubjid', 'siteid', 'sitegr1', 'racegr1', 'race', 'rfstdt',
'bmibl', 'bmibl', 'wgtbl', 'hgtbl']
   'child': ['dm.usubjid', 'dm.siteid', 'siteid', 'race', 'dm.race',
'dm.rfstdtc', 'wgtbl', 'hgtbl', 'vs.vsstresn', 'vs.vsstresn']
```

Note that each element in *parent* has a corresponding element in *child*.

Then we generate a tree-shaped *dict* (*dict* is a data structure in Python used to store key: value pairs):

```
df = pd.DataFrame(
    {
        'parent_id': ['usubjid', 'siteid', 'sitegr1', 'racegr1', 'race',
'rfstdt', 'bmibl', 'bmibl', 'wgtbl', 'hgtbl'],
        'child_id': ['dm.usubjid', 'dm.siteid', 'siteid', 'race',
'dm.race', 'dm.rfstdtc', 'wgtbl', 'hgtbl', 'vs.vsstresn', 'vs.vsstresn']
    }
)
lst = json.loads(df.to_json(orient='split'))['data']
# Build a directed graph and a list of all names that have no parent
graph = {name: set() for tup in lst for name in tup}
has_parent = {name: False for tup in lst for name in tup}
for parent, child in lst:
    graph[parent].add(child)
    has_parent[child] = True
# All names that have absolutely no parent:
roots = [name for name, parents in has_parent.items() if not parents]
# traversal of the graph (doesn't care about duplicates and cycles)
def traverse(hierarchy, graph, names):
    for name in names:
        hierarchy[name] = traverse({}, graph, graph[name])
    return hierarchy
result = traverse({}, graph, roots)
```

Generated structure is as below:

```
{
   "usubjid": {
      "dm.usubjid": {}
   },
   "sitegr1": {
      "siteid": {
         "dm.siteid": {}
      }
   },
   "racegr1": {
      "race": {
         "dm.race": {}
      }
   },
   "rfstdt": {
      "dm.rfstdtc": {}
   },
   "bmibl": {
      "wgtbl": {
         "vs.vsstresn": {}
      },
      "hgtbl": {
         "vs.vsstresn": {}
      }
   }
}
```

Iterate through this dict from bottom to top, the variable order is generated.

```
['hgtbl', 'wgtbl', 'bmibl', 'rfstdt', 'race', 'ragegr1', 'siteid',
'sitegr1', 'usubjid']
```

Then iterate through above list and call each variable's *derive()* function, then each variable's derivation will be done accordingly.

CONCLUSION

This paper presents an automation method when doing ADaM programming and illustrates with CNTRYNAM and FUEDT variable of ADSL dataset. The process shows that this method raises the level of programming automation.

REFERENCES

Real Python “Object-Oriented Programming (OOP) in Python 3”. Available at https://realpython.com/python3-object-oriented-programming/

ACKNOWLEDGMENTS

Great thanks to my colleague Benjamin Chiang and Howard Tien for their extremely useful advice.

CONTACT INFORMATION

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

Dan Li
Sanofi-Aventis
dan8.li@sanofi.com

If you are interested in the Python codes of this paper, please ask the author to provide.