PharmaSUG 2020 - Paper AP-255

Python-izing the SAS Programmer 2: Objects, Data Processing, and XML

Mike Molter, PRA Health Sciences

ABSTRACT

As a long-time SAS programmer curious about what other languages have to offer, I cannot deny that the leap from SAS to the object-oriented world is not a small one to be taken lightly. Anyone looking for superficial differences in syntax and keywords will soon see that something more fundamental is at play. Have no fear though, for languages such as Python have plenty of similarities to give the SAS programmer a strong base of knowledge from which to start their education. In this sequel to an earlier paper I wrote, we will explore Python approaches to programming tasks common in our industry, taking every opportunity to expose their similarities to SAS approaches. After an introduction to objects, we'll see the many ways that Python can manipulate data, all of which will look familiar to SAS programmers. With a solid working knowledge of objects, we'll then see how easy object-oriented programming makes the generation of common industry XML. This paper is intended for SAS programmers of all levels with a curiosity about, and an open mind to something slightly beyond our everyday world.

INTRODUCTION

If there's one thing I've learned from my enthusiasm for astronomy and cosmology, it's that the universe is a strange place. But if there's two things I've learned, then the second is that "strange" is in the eye of the beholder. After all, why should the fact that the velocity and location of microscopic particles at the quantum level are governed by probability laws, or that time slows down in the presence of intense gravity, be considered strange if they're common phenomena across the universe? Most of us know only what we see and what life experience has taught us. When we live in that bubble, it is only the most open of minds that can appreciate truly universal nature, if not fully understand it.

While the SAS bubble is one that has been good to most of us, allowing us to make careers and support families with it, the fact is that it is still a bubble. And the longer one has spent in it, the greater the challenge it is to contemplate anything else. In this bubble, nothing is more natural than processing data with a programming construct – the data step - that takes advantage of an iterative cycle that processes one record at a time, one after another. Just as it's hard to picture more than three spacial dimensions – a requirement of string theory – we might be hard-pressed to imagine a need for variable types outside of character and numeric. And yet the object-oriented world, and Python in particular, offers data processing approaches we never would've thought of before, along with data types you couldn't imagine. Your biggest challenge is to put yourself in that same mindset that allowed you to learn SAS in the first place, to clear your mind of any roadblocks the SAS bubble might be throwing at you, and to start anew with an open mind.

If SAS and object-oriented programming are two different bubbles, or dare I say, two different universes, then our best bet is to find some connection between them. Since the 1930s, physicists have contemplated the existence of wormholes – shortcuts that connect two distant points in spacetime, or possibly two different universes. In this paper we will start by finding that wormhole – what is it that connects these two seemingly different worlds. Once discovered, we'll make our way through that wormhole to see what we find on the other side. After we've looked around a bit, we'll explore how they process data on the other side. We'll also see the production of XML as an example of something made easier when we visit other worlds.

FINDING THE WORMHOLE

With SAS being such a data-centric language, let's see if we can find signs of a wormhole by making some obvious, but important observations about the SAS data set. The data set is at the center of almost everything we do in SAS, and the data step is the main tool we have for interacting with it. If we aren't directly extracting and manipulating data, we're analyzing or reporting it with PROCs. We might also be working with the macro facility in an effort to generate data-centric code. Of course data sets convey

multiple pieces of information about observations through data set variables. In SAS, these data sets are physical, proprietary files you can see in your file manager program and attach to emails.

While Python has the ability to process data (much more on this later), almost none of the rest of what we said about the role of data in the SAS programming language is true about Python data processing. Nobody would describe Python as a data-centric language, and while we can construct two-dimensional sets of data made of rows and columns, such data sets are not the basis for everything we do in Python. They do not exist as physical files on your server, but rather, as places in memory we call *objects*. It appears that our first candidate for a wormhole – the data set – is a swing and a miss.

Let's make a few more obvious observations about SAS. In addition to the data set, SAS has a few more "things" to help us in our processing of data. SAS has something called a *format* for mapping data values to other values. SAS has *templates* for designing output. Formats and templates are stored in containers like catalogs and template stores that, like data sets, are physical files. The data step has something called an *array* – a tool that expedites common processing across multiple variables. The SAS syntax contains syntax we might informally call a *list*, which might serve as the object of an *in* operator (e.g. if var in ('a', 'b', 'c')) or the basis of iterative processing (e.g. do j = 'a', 'b', 'c'). In SAS, while all of these things ultimately serve purposes centered around data, they are all otherwise independent of each other with little else in common.

On the other hand, while Python contains much of the same functionality, everything in Python is an *object*. While on the surface, the word "object" may not seem any more descriptive than the word "thing" that we have been informally using to describe different SAS features, in an object-oriented world, "object" has a clear definition that gives all such features something in common. What SAS programmers think of as data set variable types – numbers and characters – are objects. Python's answer to SAS's data set is an object Python calls the *DataFrame. Dictionaries* are objects that work like SAS user-defined formats, and *Lists* are Python objects similar to the informal SAS lists. Files on your server as well as elements of an XML tree can also be turned into Python objects. Numbers, Strings, DataFrames, Dictionaries, Lists, Files, XML Elements, and countless more Python objects, while serving different purposes, all share in common the defining features of a formal object. We'll return to those details soon. For now, one more fact about Python objects might give us a clue where this wormhole might be hiding. In Python coding, references to objects are made through *variables*.

While the idea that variables are used to reference *any* Python objects, including data sets, files, and XML elements, might seem scary, let's begin by inching our way into the wormhole with something more familiar.

Example 1

SAS data step:

```
a = 3 ;
b = 'hello world' ;
c = b ;
d = length(b) ;
e = vtype(a) ;
```

SAS macro:

```
%let a = 3 ;
%let b = hello world ;
%let c = &b ;
%let d = %length(&b) ;
```

Python:

```
a = 3
b = 'hello world'
c = b
d = len(b)
e = type(a)
```

We start by observing in each environment the creation of variables a and b and the assignment of constants to each of them. As we know, the presence of quotation marks around the constant in the data step points to a profound difference between data set variables and macro variables. In the data step, SAS interprets this as the creation of a character data set variable. Because macro variables are used primarily as text substitution, no need exists for them to have a property like *type*. While functions like %eval exist to allow us to do math on macro variable values that look like numbers, in almost every way they act like data set character variables. We could enclose the value of the macro variable b in quotation marks, but rather than indicate a different type, those quotation marks would simply be part of the variable's text value.

In this respect, Python looks much more like SAS's data set variables than it does macro variables. Python sees the quotation marks and knows to interpret b as a character variable. But let's be more precise by getting our vocabulary right. Yes, b is a variable, but because Python variables are references to objects, we often refer to b as an object. More specifically, b is a *string* object.

In a different respect, Python looks much more like the macro facility. After all, the variables created in the data step are tied to data. We can't make reference to these variables outside of any context defined by the data set to which they belong. We know very well that this isn't the case with macro variables. While macro variables can be created from data values (i.e. call symput), they often are initialized outside of any data context, through macro parameters, system settings, %let, etc. Python doesn't have anything like a data step. Like macro variables, Python objects are driven less by data and more by the environment.

Like SAS, Python objects can also be initialized by references to other objects and functions of them. We know that the *type* property of data set variables like c and d depends on the variables being referenced and the functions being applied. We know that while b is a character variable, the nature of the *length* function means that d is numeric. The same principal applies to the Python object d. The *len* function takes a string object as its input and *returns* an *integer* object.

The *len* function is known in Python as a *built-in* function. Other built-in functions include *abs* (absolute value), *min*, and *max*. These work similar to their counterparts in the SAS world. Python has other familiar functions not considered built-in, called *methods*. Before we look at these, we first must return to objects for a more formal definition.

CLASSES – OBJECTS' BIG BANG

Contemporary cosmological speculation states that all matter and time was once compressed into a tiny singularity, and through an event we call "the big bang", our universe, and all of its objects, came into existence. Physicists believe that our universe looks the way it does – the four forces of nature, atomic structure, etc, - because of the delicate balance of initial conditions at the time of the big bang. Just as the matter and the everyday objects we see around us are the remnants of this primordial explosion, Python objects like strings, integers, and others are the offspring of different *classes,* each of which defines its own set of initial conditions.

Earlier we used the word "things" to refer to different SAS features – data sets, formats, templates, etc. We also said that everything in Python is an object. In SAS, we think of "character" and "numeric" as the only two values of a data sets variable's *type* property. A data set, a format, an array are all different "things" altogether. In Python though, strings, integers, lists, and DataFrames are all different *types* of objects. Strings and integers are no more alike than strings and DataFrames. An object's type is determined by the *class* from which it was constructed. A class is a code bundle that serves as a unique blueprint for how objects are created from it. Classes are uniquely defined by two features – the attributes and the methods they define. Methods are nothing more than functions defined as part of a class definition that can be applied to objects of the class.

In general, objects are created by instantiating a class. More technically, this is achieved by calling one of a class's *constructors*, a special method used for object creation. As we saw above, for built-in types such as integers and strings, Python offers shortcuts such as the assignment of constants and the application of functions.

We won't dive very deeply into class definitions in this paper, but let's take a moment here to mention their importance in the open source world. Python comes with built-in classes, but Python programmers can also build their own classes. We could make some comparisons to SAS macros – SAS has built-in macros but programmers create their own and can distribute them among their peers in the form of programs or catalogs, or even share them online. The nature of open source, however, takes this idea to a new level. In an open source community, Python programmers can define their own classes. Through code repositories such as Git, they can invite other users to review code, add features, and make edits. Version control capabilities of such repositories make it easy to manage versions and branches without anyone's code conflicting with others. Python has tools that make it easy to install such classes into their own environment, so that these types of classes appear to be part of the main Python installation. Such is the case with DataFrames. While data sets are a core part of the SAS language, DataFrames are objects defined in a class built by a Python community. As we'll see later, the familiar ways in which we work with data sets (e.g. merging, sorting) are implemented as methods.

Example 2:

SAS data step:

```
b = 'hello world' ;
bs = strip(b) ;
bu = upcase(b) ;
bf = find(b, 'world') ;
```

Python:

```
b = 'hello world'
bs = b.strip()
bu = b.upper()
bf = b.find('world')
```

Earlier we saw built-in Python functions serving purposes similar to familiar SAS functions, such as *len.* Here we see more SAS function counterparts, this time as string methods. The syntax for method application is slightly different from that of built-in functions. The use of a method requires the name of the object along with a dot to precede the name of the method. Parentheses, even without parameters, are required for Python to know that a method is being called.

Let's return briefly to Example 1. The application of SAS's *vtype* function to create the variable e results in the value "N". In Python, *type* goes beyond just character and numeric data. The concept of type refers to the class from which the object originated. Python's *type* function returns "integer" for the variable e, and would return "string" if applied to b, just as *vtype* returns "N" and "C". Unlike *vtype* though, Python's *type* can be applied to *any* object. In addition to identifying numbers and characters, *type* also identifies DataFrame objects, Element objects, and other types of objects.

SAS programmers all have their favorite and most frequently used functions. We use some of them so much that we may not give much thought to the fact that each one acts on, or is applied to either a character or numeric variable. The same idea – that a function can only be applied to a certain type of object – applies in Python too, and might be re-stated to say that methods can only be applied to objects created from the class in which the method was defined. When we define our own class, we define what we can "do to" the objects created from it. We're also accustomed to the fact that SAS functions can return either a character or numeric value. Just as SAS's *upcase* function applies to a character and returns a character, Python's *upper* function acts on and returns a string. Just as with SAS, Python's *find* function acts on a string and returns an integer. With so many more object types in Python than data types in SAS, we must always pay close attention, when learning and using a new method, to what object type it returns.

DEEPER INTO THE WORMHOLE

It appears that we have discovered the wormhole that connects our familiar SAS world with the strange world of objects and methods. As we have cautiously proceeded, at first, what we saw was familiar – characters (or what they call in the new world, strings), and numerics (broken down into integers, floats,

and others in the new world), as well as variables and functions. As we continue on our journey, the objects still look familiar, but there's something different about them.

LISTS

Up until now, we've seen objects comparable to SAS data types. We now look at another Python object whose familiarity is rooted not in data types, but in different contexts of SAS syntax. We do so by observing what each of the following SAS statements have in common:

```
do i = 'a', 'b', 'c' ; (1)
if x in ('a','b','c') ; (2)
array myarray {3} var1 var2 var3 ; (3)
%let mylist = a b c ; (4)
```

In each of the cases above, the SAS programmer is making use of an informal concept of a list, albeit in different ways, with slightly different context. (1) and (3) use lists for iteration, although the list in (3) is a list of variables, whereas in (1) they are variable values. (4) is a common technique for macro programmers to iterate through text strings. (2) uses a list for membership inclusion. What is familiar, but different in this wormhole is that Python addresses all of this same functionality with another type of built-in object called the *list*. In SAS, we can't create a variable that facilitates the membership inclusion of (2). While we can create variables in the data step or macro language through which we can iterate, at best, such variables are characters that we can hope to parse with character functions. In Python, list objects come with operations and methods that easily address everything we need.

Consider the following list object:

```
myList = ['a','b','c']
```

We first note that just as both SAS and Python interpret a leading quotation mark as an indicator of a character variable, the presence of an opening square bracket tells Python that myList is a variable referencing a list object. This object supports membership inclusion…

```
if x in myList:
```

… as well as iteration:

```
for x in myList:
   print ('hello' + x)
```

We can capture an item from a list with syntax similar to that used to capture the value of a variable named in a SAS array. Several other operations exist too, some of which are illustrated in Example 3 below.

Example 3

`myList[0]` returns 'a'
`myList + ['d','e']` returns ['a','b','c','d','e']
`myList[1:3]` returns ['b','c']

The list class also defines several methods for basic operations such as the addition, removal, and sorting of items.

```
myList.append('d')
mylist.remove('a')
mylist.sort()
```

DICTIONARIES

When a SAS user needs to create an association between data values and mapped values, she doesn't define this association with a new variable, an array, a data set, or an iterable list. Rather, she resorts to a procedure called FORMAT that creates an entry in a physical file called a *catalog*. The Python programmer creates another type of object – the *dictionary.*

```
myDict = {'a':1,'b':2,'c':3}
```

The presence of a leading open brace tells Python that a dictionary object is being created. Informally, it can help to see a dictionary as an indexed list, with values indexed by keys. Just as items of a list were captured with a numeric index (where 0 is the first item of the list), items of a values are captured with their corresponding key. In this case, myDict['b'] returns 2.

Adding to a dictionary is as simple as an assignment statement:

```
myDict['d'] = 4
```

We can iterate through the keys with a single iterator:

```
for x in myDict:
  print (x)
```

The above prints the values 'a', 'b', 'c', 'd'. The *items* method returns a list of key-value pairs (such pairs are objects called *tuples* and are similar to lists). We can use two iterators to return keys and values:

```
for x,y in myDict.items():
  print (x,y)
```

JSON

It's important at this point to keep in mind that the items of a Python list object can be any type of object, including dictionaries and lists. The value part of a key-value pair in a Python dictionary can also be another dictionary or a list, or again, other types of objects. These facts about lists and dictionaries are key to Python's ability to process JSON.

JSON, short for Javascript Object Notation, is a lightweight, text-based structure used for data transportation. We bring it up here because the Python dictionaries we've been discussing are what Javascript calls "objects". So whether creating JSON inline in a Python program, reading it from a text file, or returning it from the internet, processing JSON and extracting data from it for Python is nothing more than working with a dictionary.

An example of JSON-formatted data can be seen in Figure 1 below. This was obtained as a response to a request made from the CDISC library and represents certain library metadata about the STUDYID variable in the VS doman. Let's now see examples of how to extract information from this complex dictionary.

```
{
  "ordinal": "1",
  "name": "STUDYID",
  "label": "Study Identifier",
  "description": "Unique identifier for a study.",
  "role": "Identifier",
  "simpleDatatype": "Char",
  "core": "Req",
  "_links": {
    "self": {
      "href": "/mdr/sdtmig/3-2/datasets/VS/variables/STUDYID",
      "title": "Study Identifier",
      "type": "SDTM Dataset Variable"
    },
    "modelClassVariable": {
      "href": "/mdr/sdtm/1-4/classes/GeneralObservations/variables/STUDYID",
      "title": "Study Identifier",
      "type": "Class Variable"
    },
    "parentProduct": {
      "href": "/mdr/sdtmig/3-2",
      "title": "Study Data Tabulation Model Implementation Guide: Human Clinical T
      "type": "Implementation Guide"
    },
    "parentDataset": {
      "href": "/mdr/sdtmig/3-2/datasets/VS",
      "title": "Vital Signs",
      "type": "SDTM Dataset"
    },
    "rootItem": {
      "href": "/mdr/root/sdtmig/datasets/VS/variables/STUDYID",
      "title": "Version-agnostic anchor resource for SDTMIG variable VS.STUDYID",
      "type": "Root Data Element"
    },
    "priorVersion": {
      "href": "/mdr/sdtmig/3-1-3/datasets/VS/variables/STUDYID",
      "title": "Study Identifier",
      "type": "SDTM Dataset Variable"
    }
  }
}
```


Figure 1

We first note that the text in Figure 1 begins with an open curly brace and ends with a closed one. That means that everything illustrated can be considered one dictionary object that we'll reference with the variable myStudyID.

Among the keys in this dictionary are 'name', 'label', 'description', 'role', and 'core' – properties we're accustomed to seeing in the SDTM IG. myStudyID['name'], myStudyID['label'], etc, return values expected of these properties.

Another key is '_links', which, unlike the keys we just mentioned, is mapped to another dictionary. In other words, myStudyID['_links'] returns another dictionary with its own set of key-value pairs. Each of these keys, such as parentProduct and parentDataset, is mapped to yet another dictionary. Putting it all together, the string "SDTM Dataset" can be captured with the following reference:

```
myStudyID['_links']['parentDataset']['type']
```

NOW IT'S GETTING WEIRD

When we first ventured into the wormhole, we found objects we could relate to. Though SAS people don't call them objects, we could get onboard with objects of numeric and character type, even if we used the term "string" instead of "character", and broke down numeric into finer distinctions such as "integer" and "float". Though we don't use terms like "method" much, we could appreciate the idea of functions taking in objects of one type and returning objects of the same type or another type. But as we got deeper into the wormhole, we saw that the word "object", or the more familiar term "type", take on more

than what SAS's data type encapsulates. SAS functionality beyond the data type of a data set variable, like lists and formats, fall under the same object umbrella in a language like Python that integers, floats, and strings do. This means that Python functions don't have to accept and return just numeric or character objects, but any other kind of object as well. For example, the string method *split* takes a delimited string and returns a list. The following returns the list ['a', 'b', 'c']:

```
myString = 'a,b,c'
myString.split(sep=',')
```

This trend of turning everything in SAS into objects continues. We're now at the point in that old Star Trek or Twilight Zone episode when we see someone familiar, we starting talking to them, but it's clear that while they look the same and sound the same, they're very different from what we knew. Not only can something that looks like a SAS format or list become an object referenced by a variable, but the same can be said about something that looks like SAS's fundamental unit of data itself – the data set.

DATA PROCESSING WITH PANDAS

Before we dive into that, let's take a moment and re-visit an idea we mentioned briefly earlier. Python is generally divided between its core functionality (standard library) that comes with the installation, and third-party functionality that does not get installed with the standard library, but rather must be downloaded and installed separately. In the latter case, such downloads come in the form of a *package.* A package contains one or more *modules*, each of which defines a combination of functions, attributes, and classes. The most popular data processing Python package is *Pandas.* The two main objects coming out of this package are the *Dataframe* and the *Series*. We'll spend most of our time on the Dataframe, but it will help to know something about the Series too. Once installed, the following statement gives your program access to the pandas package:

```
import pandas
```

A Series object is much like a dictionary object in the sense that it can be thought of as having key-value pairs. However, we find it more useful to talk about a Series as a one-dimensional "vector" of objects, where each object is labeled. The labels, or the keys are referred to as the Series' *index.* Recall that with simpler objects, we could use the presence of certain characters such as quotation marks, square brackets, and curly braces as a shorthand for creating certain object types with constant values. To create a constant value for a Series object, we pass parameters to the Series *constructor*, a special function named for the class used to create objects from a class. Since a Series looks so much like a dictionary, we can create a Series object simply by passing a dictionary to the Series constructor:

```
mySeries = pandas.Series({'a':1, 'b':2, 'c':3})
```

Note that because we were using a function from a different package, we were required to include the name of the package in our function call, separated by a dot.

Alternatively, we can also pass a list to the constructor:

```
mySeries = pandas.Series([1,2,3],index = ['a', 'b', 'c'])
```

Here we use the index parameter to pass the index, a list object itself, along with the data list. We can also pass a list without an index, in which case an index with entries 0…(n-1) will be created, where n is the number of objects in the passed list.

Because a Series object is so much like a Dictionary object, we can extract data from a Series with the same syntax we use for dictionaries. The following returns a value of 3:

```
mySeries['c']
```

An important difference between a dictionary and a Series is that binary operations (equivalent to mathematical vector operations) are defined for Series'. For example, the sum of two Series' results in a third Series whose index is the union of the indexes of the first two, and the value corresponding to a particular index is the sum of the entries from the first two with the same index. Several methods also exist for Boolean comparisons of two Series'. The following returns True if each object of the Series s1 is the same as its corresponding object (by index) of s2:

```
s1.eq(s2)
```

A DataFrame object, on the surface, mirrors the two-dimensional SAS data set. However, whereas a data set variable cannot exist without a data set, a DataFrame object can and should be thought of as a collection of Series objects with like indexes. What this means is that unlike a data set but like a Series, every DataFrame has an index. Just as the index labels a single object for a Series, it labels a row in a DataFrame. In the data step we reference data values in a column by referencing the label of that column (the variable name), but we identify rows with values of key variables. In Pandas, we identify rows the same way we identify columns – with labels. Because of this, some methods treat rows as Series objects just as columns are, and application of such a method makes use of an *axis* parameter to specify whether the operation should be performed on rows or columns.

Like Series, DataFrame objects can be created with a constructor. The following example creates a DataFrame by passing a list of dictionaries, each with the same key, as a parameter:

```
DM =
pd.DataFrame([{'USUBJID':'001','SITEID':'ABC'},{'USUBJID':'002','SITEID':'DEF
'},{'USUBJID':'003','SITEID':'ABC'}])
```

As a two-dimensional data structure, in SAS we would see this example as a data set with three observations and two variables – USUBJID and SITEID. In pandas, this is a DataFrame that contains a Series named USUBJID and another Series named SITEID.

If a DataFrame is the two-dimensional collection of one-dimensional Series objects, then it makes sense that the selection of data extends the way we select data in a Series. Just as mySeries[‘c’] returned a single zero-dimensional integer, DM[‘USUBJID’] returns a one-dimensional Series. In addition, by referencing a list of Series names in the brackets rather than a string of one Series name, the following returns another two-dimensional DataFrame containing only the Series objects named in the passed list.

```
VS[[‘STUDYID’, ‘VSTESTCD’, ‘VSORRES’]]
```

The outer brackets are part of the selection syntax, while the inner brackets denote a list. This works exactly like the KEEP= data set option or KEEP statement in the data step.

Pandas Methods

Different operations on SAS data sets require different syntax to carry them out. Sorting a data set requires the execution of a SAS procedure; dropping, keeping, and renaming variables can be done with data set options or statements; merging two data sets requires a data step statement. With Pandas, many such operations are accomplished with methods. The following examples illustrate some of these methods applied to a DataFrame called df.

```
df.drop([‘b’, ‘c’],axis=1)
```

Here we are passing list and integer objects into parameters. The list object is a list of the labels of the columns to be dropped. The *axis* parameter tells Pandas that these are, in fact, column labels and not row labels. The default value of *axis* is 0, which refers to row labels.

```
df.rename({‘a’ : ‘a1’, ‘b’ : ‘b1’},axis=1)
```

The rename function takes a dictionary whose keys are the original labels, and whose corresponding values are the new labels. Once again, the value of the *axis* parameter indicates that the labels are column labels. In this case, column labels ‘a’ and ‘b’ are changed to ‘a1’ and ‘b1’ respectively.

Let’s now suppose that AE is a DataFrame object. Can you guess what the following does to AE?

```
AE.sort_values([‘USUBJID’, ‘AETERM’],ascending=[True,False],
na_position=’first’)
```

The *sort_values* method takes a list of strings, each of which represents labels, and sorts the DataFrame, first by USUBJID, then by descending values of AETERM. The BY statement in PROC SORT takes a space-delimited “list” of variable names, each of which can optionally be preceded by the word “descending”. Here, the list of string objects that represent column labels comes by way of a list object.

By default, all sorting is done by ascending order, but if any columns are to be sorted in descending order, then this has to be passed by way of the *ascending* parameter that takes a list of Boolean objects (a class whose only two objects are True and False), the nth of which applies to the nth string object in the passed list of labels. By default, missing values, unlike in SAS, are sorted to the bottom, but the *na_position* parameter takes string values of 'first' and 'last', allowing us to decide how such values are sorted.

```
AE.drop_duplicates(subset=['USUBJID', 'AETERM'],keep='last')
```

Pandas also has something like a *nodupkey*, but it's accomplished with a different method – drop_duplicates – and has nothing to do with sorting. Without providing any arguments, this method simply looks for records that are duplicated in all variable values and removes all such records except one. The *subset* parameter allows you to specify a subset of Series objects in which to look for duplicates (instead of all variables). These are the variables that would be found on your BY statement in a PROC SORT. In contrast to *nodupkey* which keeps only the first of multiple records with the same BY values, the *keep* parameter allows you to indicate which record to keep. Allowable values are "first" (default), "last", or the Boolean False, which removes all such records.

The following merges two DataFrame objects:

```
AE2 = AE.merge(suppae,how='left', on='usubjid', indicator=True)
```

In SAS, this might look something like the following:

```
data ae2 ;
merge AE(in=inAE) suppae(in=insupp) ;
by usubjid ;
if inAE ;
```

The *how* parameters makes sure that every record in AE is kept. Other allowable values for this parameter are 'right' (every record in SUPPAE is kept), 'inner' (only matching records are kept), and 'outer' (records from either DataFrame are kept). The *on* parameter can be a string if only one BY variable is needed, or a list of strings if more are needed. The INDICATOR adds a column to the output DataFrame called *_merge,* whose allowable values are 'both', 'left_only', and 'right_only' to indicate from which DataFrame(s) the record came.

Pandas offers a watered-down version of SAS's PROC TRANSPOSE. In the following case, values of QNAM become column headers while values of QVAL become data values. USUBJID becomes an index (e.g. values become row labels) in the output DataFrame.

```
suppdm2 = suppdm.pivot(index='usubjid', columns='qnam', values='qval')
```

In short, what we've seen up until now is that familiar SAS functionality available through SAS statements, procedures, data step statements, syntax, etc. is oftentimes implemented through methods in Python. In many of these cases, where SAS requires a list (e.g. space-delimited list of variables in the BY statement of PROC SORT), Python methods require a list-populated method parameter. Another place we see lists in SAS syntax is in PROC MEANS/SUMMARY/UNIVARIATE – class variables, requested statistics, analyzed variables. This can be accomplished by passing lists or dictionaries to the AGG method.

```
advs[['aval', 'chg']].agg(['sum', 'mean'])
advs[['aval', 'chg']].agg({'aval' : 'sum', 'chg':['sum', 'mean']})
```

The first of these computes the sum and mean of AVAL and CHG across all records and returns a DataFrame composed of a Series named AVAL and another named CHG. The indexes (or row labels) are the statistics – sum and mean. The second computes the sum of both AVAL and CHG as well as the mean of CHG, again returning a DataFrame indexed by the statistics names.

In the case above, we computed the statistics over all records. More typically we want to compute them once per group of records. In the SAS PROCs, the CLASS statement defines the groups (as well as the BY statement). In Pandas, we first need to define a new object – a *GroupBy* object. If LBTESTCD is the SAS CLASS variable, then we create a GroupBy object as follows:

```
grpobj=LB.groupby('LBTESTCD')
```

… or LBTESTCD and VISITNUM:

```
grpobj2=LB.groupby(['LBTESTCD', 'VISITNUM'])
```

We can iterate through such an object with two iterators, the first of which returns the name of the group, or *key* (a string object when a string was passed to the groupby object, and a tuple object when a list was passed). The second is a DataFrame object associated with the corresponding key. We can also capture the data (DataFrame) corresponding to a given key with the *get_group* method:

```
grpobj.get_group('ALT')
```

We can also create summaries by group using the *grpobj* object and the *agg* method, similar to above.

```
grpobj['LBSTRESN'].agg(['sum', 'mean'])
```

Other Pandas Functionality

Many ways exist to filter a Pandas DataFrame, but here we'll focus on an approach called *Boolean Indexing*. For a DataFrame called df, the syntax is as follows:

```
df[condition]
```

`condition` is what is referred to as a Boolean vector. We can think of it as a Series object with as many entries as df has rows, and the values of the vector are True or False. Consider the following:

```
df[df['CodeListCode'] == 'C141663']
```

The expression `df['CodeListCode'] == 'C141663'` represents the Boolean vector with values of True and False. Passing this Series object inside of square brackets selects rows in the same way that passing a list of Series names selects columns.

The DataFrame counterpart of creating a new variable in a data set is the creation of a new Series in the DataFrame. This is where SAS programmers need to be careful. Consider the following:

```
df['CHG'] = df['AVAL'] – df['BASE']
```

This works in a DataFrame the same way that CHG = AVAL – BASE would work in a SAS data step, but not for the same reasons. Keep in mind that the data step is an iterative process, working on one row at a time. So a reference to AVAL (as well as to BASE) is a reference to a single value of a variable. Subtraction of two single values is well-defined, resulting in a single value for CHG. In the DataFrame, however, df['CHG'] is a Series object created from the subtraction of two Series objects. Luckily, subtraction is an operation that is defined on two Series objects, and is carried out by subtracting corresponding entries of the two objects (where entries are corresponding if they have the same index values).

Let's suppose, however, that we only want to calculate CHG when the value of another column, POSTBASE, is 1. Something like the following would certainly be tempting:

```
if df['POSTBASE'] == 1:
   df['CHG'] = df['AVAL']-df['BASE']
```

This is typical Python syntax for conditional logic. The problem is in the condition. Keep in mind that df['POSTBASE'] is a Series object, so to try and compare it to a value of 1 is undefined. For that reason, an error would be thrown, and we need another approach to creating a Series object within our DataFrame whose values are assigned conditionally.

Since we're creating a new Series object, recall that we can do so by passing a list to the Series constructor. Up to this point, we've only built lists by providing explicit values to it. We can also build lists by providing a formula. This approach is called *list comprehension.* For example, the following creates a list whose values are 0, 1, 4, and 9:

```
[x*x for x in range(4)]
```

The range function is a built-in Python function that, when supplied only one positive integer value, returns a sequence starting at 0 and ending at the integer before the provided argument. List comprehension generates a list by allowing an iterator to iterate over a sequence of values, and evaluate the expression for each value in that sequence. In our sequence, we'll iterate over the values of the POSTBASE Series object. The expression that will be evaluated will be what is referred to as a *conditional expression* – an alternative to traditional conditional logic:

```
df['CHG'] = pandas.Series(df['AVAL']-df['BASE'] if x==1 else "" for x in df['POSTBASE'])
```

This works fine if the condition is based on only one column. Suppose though that we add the condition that the value of the ONTRTFN column must also be 1. For that, we have the *iterrows* method.

iterrows iterates through each of the rows of a DataFrame and at each iteration returns two items: the index of the row, and the set of data in that row in the form of a Series. Because two items are returned, we need two iterating variables.

```
for idx,row in df.iterrows():
   <code>
```

At each iteration (each row of the DataFrame), we can capture the index of the row with a reference to *idx*. *row*, on the other hand, is a Series whose data values are the data values in that row of the DataFrame, and whose index is the set of column headers. This means that in each row, we can capture values of any of the columns. We'll use list comprehension as above, but this time we'll iterate not through the values of a single column, but through the rows:

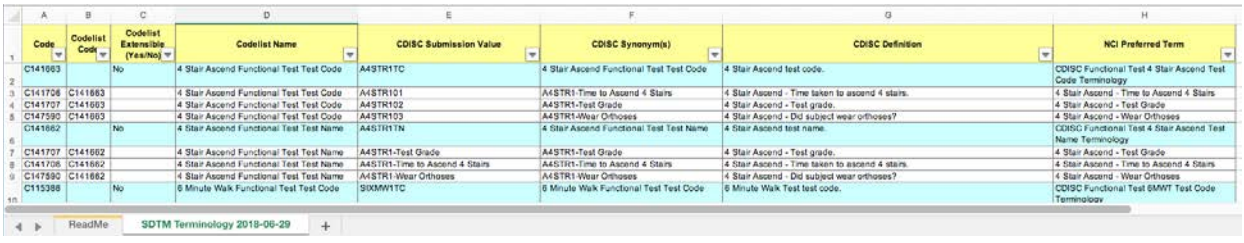
```
df['CHG'] = pandas.Series(df['AVAL']-df['BASE'] if (row['ONTRTFN']==1) & (row['POSTBASE']==1) for idx,row in df.iterrows())
```

An Exercise – Part 1

We're now ready to put some of what we've learned here to use in an exercise, the purpose of which might be near and dear to your heart if you have any need to process controlled terminology. Part 1 of this exercise will involve reading one of the NCI controlled terminology spreadsheets into a DataFrame and calling methods on it in preparation for Part 2 of the exercise, writing it to XML. We'll go over the tools we need for Part 2 in the next section.

Please note that this exercise is only an exercise in manipulating DataFrame data and writing XML with Python, and should in no way be interpreted as instructions for turning the NCI spreadsheet into define.xml, or even if such a task should be attempted.

Figure 2 below illustrates a small portion of the spreadsheet we'll be reading in.



	A	B	C	D	E	F	G	H
1	Code	Codelist Code	Codelist Extensible (Yes/No)	Codelist Name	CDISC Submission Value	CDISC Synonym(s)	CDISC Definition	NCI Preferred Term
2	C141663		No	4 Stair Ascend Functional Test Test Code	A4STR1TC	4 Stair Ascend Functional Test Test Code	4 Stair Ascend test code.	CDISC Functional Test 4 Stair Ascend Test Code Terminology
3	C141706	C141663		4 Stair Ascend Functional Test Test Code	A4STR101	A4STR1-Time to Ascend 4 Stairs	4 Stair Ascend - Time taken to ascend 4 stairs.	4 Stair Ascend - Time to Ascend 4 Stairs
4	C141707	C141663		4 Stair Ascend Functional Test Test Code	A4STR102	A4STR1-Test Grade	4 Stair Ascend - Test grade.	4 Stair Ascend - Test Grade
5	C147590	C141663		4 Stair Ascend Functional Test Test Code	A4STR103	A4STR1-Wear Orthoses	4 Stair Ascend - Did subject wear orthoses?	4 Stair Ascend - Wear Orthoses
6	C141662		No	4 Stair Ascend Functional Test Test Name	A4STR1TN	4 Stair Ascend Functional Test Test Name	4 Stair Ascend test name.	CDISC Functional Test 4 Stair Ascend Test Name Terminology
7	C141707	C141662		4 Stair Ascend Functional Test Test Name	A4STR1-Test Grade	A4STR1-Test Grade	4 Stair Ascend - Test grade.	4 Stair Ascend - Test Grade
8	C141706	C141662		4 Stair Ascend Functional Test Test Name	A4STR1-Time to Ascend 4 Stairs	A4STR1-Time to Ascend 4 Stairs	4 Stair Ascend - Time taken to ascend 4 stairs.	4 Stair Ascend - Time to Ascend 4 Stairs
9	C147590	C141662		4 Stair Ascend Functional Test Test Name	A4STR1-Wear Orthoses	A4STR1-Wear Orthoses	4 Stair Ascend - Did subject wear orthoses?	4 Stair Ascend - Wear Orthoses
10	C115388		No	6 Minute Walk Functional Test Test Code	SIXMW1TC	6 Minute Walk Functional Test Test Code	6 Minute Walk Test test code.	CDISC Functional Test 6MWT Test Code Terminology

ReadMe | SDTM Terminology 2018-06-29 | +

Figure 2

The following illustrates the XML format with a portion of the AESEV codelist.

```
<CodeList Name="Severity/Intensity Scale for …">
   <CodeListItem CodedValue="MILD">
      <Decode>
         <TranslatedText>Grade 1</TranslatedText>
      </Decode>
      <Alias Name="C41338"/>
   </CodeListItem>
```

```
Other CodeListItem elements formatted as above, one for each term

      <Alias Name="C66769"/>
   </CodeList>
```

If you know this spreadsheet, then you know that codelist-level information is found on the blue lines and codelist item-level information is found on the white. In the XML, this means that each white row gets its own CodeListItem element which is nested within the CodeList element represented by the blue row above it. CodeList-level information is found inside the CodeList element but outside of any CodeListItem element.

With the structure of the Excel source and the structure of the XML target under consideration, a SAS programmer planning to write XML with a data step and PUT statements immediately begins thinking about the necessary data step code to get from source to a structure from which PUT statements are straightforward. While different programmers might choose different approaches and structures, we'll pick one here, summarize it, and then see how we can accomplish the same steps in Python.

In short, we'll take the approach of importing the spreadsheet into one data set, separate it into two data sets, each with different variables, where one represents the headers (blue rows, column B is missing) and the other represents the details (white rows, column B is not missing), and then merge them back together by a common key, so that the number of rows in the final data set matches the number of white rows in the spreadsheet, with codelist-level information repeated. This information, which need only be printed in the XML once per codelist, can then be generated using "first-dot" and "last-dot" logic.

SAS gives us several different methods for importing data from several different kinds of sources. PROCs such as PROC IMPORT allow us to read from Excel or CSV. DDE is an older alternative. The data step with the INPUT statement lets us read flat files. The XML engine and XML maps let us use the LIBNAME statement to read XML. Pandas provides methods for reading Excel, CSV, JSON, and even SAS files. This means that we can create a DataFrame object from, for example, an Excel file, with something that looks like a parameterized SAS macro call.

```
df = pandas.read_excel(parameters)
```

The *read_excel* method parameters allow you to specify the Excel file as well as the sheets from which data is to be read, which columns and rows are to be read, where (if any) the header rows are, and how to process missing values, dates, and thousands separators.

```
ctdf=pd.read_excel('SDTM_Terminology_2018-06-
29.xlsx',sheetname=1,usecols=[0,1,2,3,4,7],names=['Code','CodeListCode','Exte
nsible','CodeListName',SubmissionValue,'Decode'],keep_default_na=False)
```

The first parameter is positional and is the name of the file. This can also contain a path or a URL. The *sheetname* parameter defaults to the first tab in the file when not specified. Otherwise it is populated with an integer as above or a string (name of the tab). Note that while we chose the second tab, we specified "1" as the value. Almost everything in Python is zero-indexed, which means that to ask for the first of something, like a sheet in an Excel file, or an element of a list, you specify 0. This parameter can also be populated with a list of multiple sheets (integers or strings). The *usecols* parameter specifies a list of columns to process (again, zero-indexed). In this case, we left out the CDISC synonym and definition. The *names* parameter names our columns using a list. Finally, the *keep_default_na* parameter is a Boolean (notice the lack of quotation marks). By default, Pandas assumes that certain values (e.g. NA, N/A, #N/A, and others) are to be interpreted as missing values. Setting this parameter to False turns that assumption off. Users also have the option of specifying other values which are to be interpreted as missing through the *na_values* parameter (not seen here).

We now have a Pandas DataFrame object named CTDF. Let's suppose that the SAS data set counterpart – the result of importing from Excel using SAS – is called CTDS. In separating CTDS into two data sets – a header data set (HDS) and a detail data set (DDS) we can also accomplish two other necessary tasks at the same time – restrict the variables in each and rename.

If you look carefully at column A of the spreadsheet – the one we named "Code" when creating CTDF – you'll see why it needs renaming. On the header lines – the values of this column represent Codelist-level codes, but in the detail lines, they are item-level codes. For that reason, we will rename "Code" to "CodeListCode" in the header data set. Since the detail rows also have this same code in the variable we're already calling CodeListCode, we'll keep this variable in the detail-level data set so that we have something to merge on when the time comes. The following is straightforward data step code to accomplish the three tasks of separating based on a condition, selecting variables to be kept in each data set, and renaming:

```
data
   hds(keep=Code Extensible CodeListName rename=(Code=CodeListCode)
   dds(keep=Code CodeListCode SubmissionValue Decode) ;
set ctds ;
if missing(CodeListCode) then output hds ;
else output dds ;
```

Let's now build the Python code step by step. The following creates a DataFrame that simply filters on the DataFrame that was created from *read_excel*:

```
hdf = ctdf[ctdf['CodeListCode'] == '']
```

The following adds the restriction on "variable" names or Series names to keep:

```
hdf = ctdf[ctdf['CodeListCode'] == ''][['Code', 'Extensible',
'CodeListName']]
```

The following adds the rename:

```
hdf = ctdf[ctdf['CodeListCode'] == ''][['Code', 'Extensible',
'CodeListName']].rename({'Code': 'CodeListCode'},axis=1)
```

Let's now put it all together. Just as SAS allows us to create two data sets in one data step, we can also create two objects at once in Python. The following creates both the header and detail DataFrame objects:

```
hdf,ddf = ctdf[ctdf['CodeListCode'] == ''][['Code', 'Extensible',
'CodeListName']].rename({'Code':
'CodeListCode'},axis=1),ctdf[ctdf['CodeListCode'] != ''][['Code',
'CodeListCode', 'SubmissionValue', 'Decode']]
```

We now have two DataFrame objects that we will merge together into one DataFrame (FINALDF) by the common CodeListCode:

```
finaldf = hdf.merge(ddf,on=CodeListCode)
```

Figure 3 below shows a portion of the new FINALDF DataFrame object.

```
3.py
   CodeListCode Extensible          CodeListName     Code              SubmissionValue                                  Decode
0       C117745        Yes      Analysis Purpose   C98724     EXPLORATORY OUTCOME MEASURE                 Exploratory Outcome Measure
1       C117745        Yes      Analysis Purpose   C98772          PRIMARY OUTCOME MEASURE                      Primary Outcome Measure
2       C117745        Yes      Analysis Purpose   C98781        SECONDARY OUTCOME MEASURE                    Secondary Outcome Measure
3       C117744        Yes       Analysis Reason  C117750                      DATA DRIVEN                        Data Driven Analysis
4       C117744        Yes       Analysis Reason  C117751  REQUESTED BY REGULATORY AGENCY          Analysis Requested by Regulatory Agency
5       C117744        Yes       Analysis Reason  C117752           SPECIFIED IN PROTOCOL                  Analysis Specified in Protocol
6       C117744        Yes       Analysis Reason  C117753               SPECIFIED IN SAP    Analysis Specified in Statistical Analysis Plan
7        C81223         No  Date Imputation Flag   C81212                              D                                      Day Imputed
8        C81223         No  Date Imputation Flag   C81211                              M                                    Month Day Imputed
9        C81223         No  Date Imputation Flag   C81210                              Y                                 Year Month Day Imputed
10       C81224        Yes       Derivation Type   C81209                        AVERAGE           Average of Value Derivation Technique
11       C81224        Yes       Derivation Type   C92225                             BC                  Best Case Imputation Technique
12       C81224        Yes       Derivation Type   C81201                          BLOCF  Baseline Observation Carried Forward Imputatio...
13       C81224        Yes       Derivation Type  C132340                            BOC          Best Observed Case Imputation Technique
14       C81224        Yes       Derivation Type   C92226                           BOCF  Best Observation Carried Forward Imputation Te...
15       C81224        Yes       Derivation Type   C82866                       ENDPOINT              Endpoint Value Derivation Technique
16       C81224        Yes       Derivation Type  C139176                         EXTRAP               Extrapolation Imputation Technique
                                                                                                  Ln 9, Col 1  Spaces: 4  UTF-8  LF  Python  1
```

Figure 3

CREATING XML

One of the clearest illustrations of the advantages that an object-oriented language can have is in writing XML. Imagine that the SAS counterpart to FINALDF is a SAS data set called FINALDS. Because SAS

has no shortcuts to writing custom XML (e.g. a PROC or an ODS destination), SAS programmers have to write the XML the old-fashioned way – with FILE and PUT statements in the data step. Since Codelist-level metadata is repeated for every record that represents a codelist item in that codelist, "first-dot" and "last-dot" logic might be employed (after sorting, of course) to generate start and end tags. The following illustrates how this code might look:

```
data _null_ ;
set finalds ;
by codelistcode ;
if first.codelistcode then put '<CodeList Name=' + quote(codelistname) +
'>' ;
put '   <CodeListItem CodedValue=' + quote(submissionvalue)+ '>' ;
put '      <Decode>' ;
put '         <TranslatedText>'+strip(decode)+ '</TranslatedText>'
put '      </Decode>' ;
put '      <Alias Name='+quote(code)+ '/>' ;
put '   </CodeListItem>' ;
if last.codelistcode then do ;
   put '   <Alias Name='+quote(codelistcode)+ '/>' ;
   put '</CodeList>' ;
end ;
```

This code looks fairly clean and concise on the surface, but let's make a few observations regarding some of the "energy" required. For starters, the programmer is responsible for every character written. In other words, the programmer has to write each "<" and ">", "=", and quotation marks around attribute values. In order to make the XML readable, the programmer is responsible for all indentation. The programmer is also responsible for proper nesting. That means that the programmer not only generates the start tag, but they also have to generate the end tag, and they have to do so in the right spot.

Programmers also have to consider XML special entities. In other words, when reading data from a data set, if data values that are being written contain any special XML characters (e.g. "<", ">", "&", etc), the programmer must substitute the proper entities (e.g. <, >, &, etc). The HTMLENCODE function helps with that, but each substitution adds more characters. The programmer will need to be sure to have a long enough variable length to account for these. Finally, SAS writes to flat files with a logical record length. Programmers will need to account for this.

So while the code above doesn't look too daunting, we can see that significant energy is being spent on efforts that have nothing to do with the content itself, and this code can get lengthy quickly with a longer XML tree, like define.xml. On the other hand, a good object-oriented XML package takes care of most of these hassles.

By this point it should be no surprise to you that we can generate the same XML that the SAS program above creates by simply creating objects and calling methods. In particular, we will create Element and SubElement objects. In order to print or *serialize* the results, we'll call on a method with parameters that will take care of everything we had to do manually with SAS. Please note that with this exercise, we are attempting to illustrate only how to put elements and attributes together. The result will not be a complete XML document with a root element. We will also not address namespaces, or how to write to a file. For this exercise, we will only print the resulting XML.

We'll start by importing another third-party Python package. Be sure to install this package before attempting the following:

```
from lxml import etree
```

This syntax for importing is slightly different than what we saw when we imported Pandas. Here, instead of importing the whole lxml package, we are importing just one module – *etree*.

Creating an Element object is easy – we simply call its constructor, providing the tag name as the first positional parameter, followed by keyword parameters named for the attributes of the element you are creating.

```
codelist = etree.Element('CodeList', Name = 'Derivation Type')
```

SubElement objects are similar, except the tag name is now second, and the name of the Element object representing the parent element is first.

```
codelistitem = etree.SubElement(codelist, 'CodeListItem', CodedValue =
'Average')
```

It's important to remind ourselves here that *codelist* and *codelistitem* are not string objects, but rather Element and SubElement objects. The relationship between them – that *codelistitem* is nested within *codelist* – is known. Serializing what we have right now would result in the following:

```
<CodeList Name = 'Derivation Type'>
   <CodeListItem CodedValue = 'Average'/>
</CodeList>
```

The following illustrates the serializing function:

```
print (etree.tostring(cl, pretty_print=True, encoding = 'unicode'))
```

The first positional parameter of the *tostring* function is simply the name of the outermost Element object to be serialized. *pretty_print* takes care of the readability by inserting line breaks between tags and indenting child elements. Note that nowhere in our code did we have to explicitly tell the program to generate a "<" or a "/", or to move to the next line, or give any instructions for indenting. We also did not have separate lines of code to generate the start tag and end tag for the CodeList element, even though they are physically apart from each other in the output.

THE EXERCISE – PART 2

With all of this in mind, let's now generate XML from the contents of the FINALDF DataFrame object. Recall that to generate the codelist-level metadata with SAS, we needed "first-dot" and "last-dot" logic. The Pandas counterpart is to create a GroupBy object. Recall that earlier we created these objects for the purpose of summary statistics by group. We now use it for iteration, not through rows of the DataFrame, but through groups. The following creates groups defined by codelist-level metadata:

```
ctgp = finaldf.groupby(['CodeListCode', 'CodeListName'])
```

We iterate through groups with two iterators. For each group, the first iterator is the name of the group. When only one "grouping variable" (think CLASS variable) is used, this is a string object whose value is the value of the grouping variable. When two or more grouping variables are used, as in this case, then this name is a *tuple* object (similar to a list) containing the values of all the grouping variables. The second iterator is the subset of the original DataFrame object specific to the corresponding grouping variables.

```
for name, group in ctgp:
```

According to Figure 3 above, the first iteration will yield for *name* the tuple ('C117745', 'Analysis Purpose'). The subset DataFrame containing the rest of the variables and only the rows with these as the values of CodeListCode and CodeListName will be assigned to *group*. The following contains the rest of the code:

```
for name, group in ctgp:
   codelist=etree.Element('CodeList',Name=name[1])
   for x,y in group.iterrows():
         codelistitem=etree.SubElement(codelist,'CodeListItem',CodedValue=
         y['SubmissionValue'])

         decode=etree.SubElement(codelistitem,'Decode')
         trantext=etree.SubElement(decode,'TranslatedText')
         trantext.text=y['Decode']
         itemalias=etree.SubElement(codelistitem,'Alias',Name=y['Code'])
```

```
codelistalias=etree.SubElement(codelist,'Alias',Name=name[0])
print(et.tostring(codelist,pretty_print=True,encoding='unicode')
```

Before we go through this, let's make a couple observations about Python syntax. First, as you may have noticed by now, Python's *for* is similar to SAS's *do* loop. You may have also noticed by now that Python doesn't use semicolons. Instead, Python is more dependent on code lines and indenting. Note the colon that follows the *for* statement. All statements that the iteration logic is supposed to cover all need to be indented the same amount to the right of where *for* started. This means that all code that is one level of indenting to the right of the first *for* is executed once per group, but all the code indented to the right of the second *for,* which iterates through all rows of the current DataFrame subset (as we saw earlier when we first saw *iterrows*), executes once per row of that DataFrame. That is how for each group, a CodeList element is created only once, but we are able to generate a CodeListItem element within that CodeList element from each row in the subset.

Python's iterative *for* loop is different than SAS's iterative *do* loop in that it is more abstract. With the *do* loop, we can explicitly see what values the iterating variable takes on. For example, do i = 1 to 10 clearly sets the value of i to 1 on the first iteration, 2 on the second, and so on. While this kind of iterating can be done with *for* loops too, iteration though different kinds of objects isn't so obvious, and the value of the iterators depends on the type of group being iterated through. Iterating through a GroupBy object produces two objects at each iteration rather than one – the first is either a string or a tuple, depending on whether or not more than one grouping variable was specified in the GroupBy object definition, and the DataFrame subset that corresponds to the string or tuple. Iterating through the result of applying the iterrows method (i.e., iterating through the rows of a DataFrame) also produces two objects at each iteration – the row label (index), and the row itself, expressed as a Series object indexed by the column labels.

In the first iteration through the GroupBy object CTGP, the value of "name" is the tuple ('C117745', 'Analysis Purpose'), and the value of "group" is the associated subset of the FINALDF DataFrame. We can extract values from a tuple the same way we can extract them from a list. So if "name" is the name of a tuple, then name[0] extracts the first element, name[1] gives us the second, and so on. The first statement in the outer *for* loop creates an Element object called "codelist" whose tag name is "CodeList", and that contains an attribute called "Name" whose value is the second value of the tuple – "Analysis Purpose".

We then enter the inner loop which iterates through the rows of the current value of "group" (i.e. the current subset of FINALDF). Just thinking about the first iteration, x will take on the value of the row label (index). Since no row labels were explicitly defined in this case, the index simply takes on values from 0 to one less than the number of rows. On the first iteration of the inner loop, x is set to 0, and y is set to a Series representing the first row, indexed by column headers. Recalling that we capture Series values like we capture Dictionary values where the index of the Series serves as the set of keys, y['SubmissionValue'] returns the value of SubmissionValue on this first row. We can now see that "codelistitem" is an Element object representing a subelement of CodeList, with a CodeListItem attribute whose value is set to the value of SubmissionValue. We then create a Decode subelement of CodeListItem, and a TranslatedText subelement of Decode. The *text* property of the trantext object is set to the value of Decode from the DataFrame, and represents the data value of the TranslatedText element.

Note too that we are creating two different Alias elements – one at the item level and the other at the CodeList level. The latter is created after looping through all the rows that correspond to the current codelist. We then print the current codelist, turning on the *pretty_print* option for readability.

CONCLUSION

For a scientist, the need to adjust hypotheses and theories based on new discoveries is par for the course. It's the essence of the scientific method. To some the quest for knowledge being constantly interrupted by bumps in the road might seem like an exercise in futility, but to many, what we learn along the way is well worth the struggle. As SAS programmers, discovering the details of the Big Bang, or the existence of wormholes may not be in our playing field, but discovering patterns in data that sometimes reveals the efficacy and safety of new compounds is at our fingertips. And just as the introduction of the

Hubble telescope helped us find galaxies we never dreamed of, adding new tools to our toolbox may someday allow us to expand our own imaginations.

CONTACT INFORMATION <HEADING 1>

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

Mike Molter
PRA Health Sciences
919-414-7736
moltermike@prahs.com