PharmaSUG
NORTH CAROLINA • OCT. 29. 2021

Python-izing the SAS Programmer: A Brief Introduction to the World of Objects

Mike Molter
Labcorp Drug Development
October 29, 2021

Meet the Speaker

Mike Molter
Title: Associate Director of Statistical Programming and Technology Initiatives
Organization: Labcorp

In this position, Mike works with FSP clients as well as internal study teams on the development of technical tools both inside and outside of SAS. He is also involved with the development of open source initiatives and processes as well as staff training.

Mike has been involved in SAS programming since 1999, in clinical trials since 2003, and in industry data standards since 2005. He spent several years as a member of the CDISC XML Technologies team, and is a certified CDISC instructor for the define.xml class. Professional interests are centered around the use of cutting edge technologies to optimize the use of metadata throughout the lifecycle of a clinical trial.

Agenda

- What is an object?
- Examples (SAS-like)
- Where do objects come from?
- Examples (more interesting)
- Conclusion

What is an object?

In Python, *everything* is an object!

SAS-like examples

assignment statements with literals

SAS

```
/* numeric data set variable */
mynum = 3 ;

/* character data set variable */
mychar = 'three' ;
```

Python

```
# integer object
myint = 3

# float object
myfloat = 3.14

# complex object
mycomplex = 3+4j

# string object
mystr = 'three'
```

SAS-like examples

assignment statements with functions

SAS

```
/* numeric data set variable */
mychar = 'three' ;

mynum2 = index(mychar, 'h') ;
mychar2 = upcase(mychar) ;
mychar3 = strip(mychar) ;

mynum3 = mynum1 + mynum2 ;
```

Python

```
# integer object
mystr = 'three'

myint2 = mystr.index('h')
mystr2 = mystr.upper()
mystr3 = mystr.strip()

myint3 = myint1 + myint2
```

Note: Each function, regardless of language, comes with a *return type*

Basic Comparison

- Similarities
  - Numeric and character variables/objects created with assignment statements
    - literals – processor knows the type of the variable by the value being assigned
    - functions – processor knows the type by the function's return type
    - **Type – what you can do to and what kind of information you can get from variable/object values**
- Differences
  - Python numeric objects divided into three types; strings instead of characters
  - function-calling syntax
  - SAS types are restricted to data set variables, and only two are defined
  - Python objects are "out in the open" (like SAS macro variables); Python types differentiate all objects; everything is an object

Everything is an object - Lists

SAS

```
if x in ('a', 'b', 'c')
then put x ;

do x = 'a', 'b', 'c' ;
     put x ;
end ;

array vars {3} a b c ;
```

Python

```
mylist = ['a', 'b', 'c']
if x in mylist:
     print (x)

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

List methods

```
mylist = ['a', 'b', 'c', 'd']

# indexing
mylist[0] = 'a'
mylist[1] = 'b'

# slicing
mylist[0:2] = ['a', 'b']
mylist[1:3] = ['b', 'c']

mylist.append('e') = ['a', 'b', 'c', 'd', 'e']
mylist.insert(2,'e') = ['a', 'b', 'e', 'c', 'd']
mylist.remove('c') = ['a', 'b', 'd']
mylist.sort(reverse=True) = ['d', 'c', 'b', 'a']
```

Str function that returns a list

Python

```
mystr = 'hello world how are you'

mystr.split(sep = ' ') = ['hello', 'world', 'how', 'are', 'you']

mystr.split(sep = ' ')[2] = 'how'
```

SAS

```
scan(mystr, 3, ' ') = 'how'
```

Everything is an object - Dictionaries

SAS

```
proc format ;
     value myD
     1 = 'a'
     2 = 'b' ;
run ;

/* access */
put(i, myD) = 'a'
```

Python

```
myD = {8: 'a',9: 'b'}

# access
myD[1] = 'a'

# add a mapping
myD[55] = 'c'

# Test for a key in myD
x=1
if x in myD…
```

Summary (so far)

- All the “things” we find in SAS – data set variables, lists, arrays, user-defined formats, external files – all fall under the umbrella of “objects”
- Object types are differentiated by what we can do with their values and what kind of information we can get from them.
  - Some objects can have *properties*
    - accessed by *object-name.property-name*
    - for example, if z is the complex object 3 + 5j, then z.real = 3, and z.imag = 5
  - Some objects can have *methods*
    - Methods are functions tied to a particular type of object
    - called by *object-name.method-name(<parameters, if any>)*
- Some objects can be created from literals
- An object of a given type can be created from a method that returns that type

Where do objects come from?

- Objects are born from *classes*
- Classes are templates or blueprints that define a particular type of object
- Classes can define methods and set property values
- An object of any type can be created by calling a *constructor* – a special method defined by the class used for instantiating objects
  - The name of a constructor always matches the name of the class
  - `z = complex(2,4)` (the complex object 2+4j is assigned to z)
  - The complex class defines the *real* and *imag* properties (z.real = 2, z.imag=4)
  - The complex class defines methods for adding/subtracting two complex numbers, comparing for equality, and more.
- Up until now, all classes we've seen are built into the core of Python
- Others are built into *modules* that must be imported with an *import* statement
- Others are written by third-party contributors and are available for download and installation (open source)

Importing JSON


```
{
  "ordinal": "11",
  "name": "VSPOS",
  "label": "Vital Signs Position of Subject",
  "description": "Position of the subject during a measurement or examinatior
  "role": "Record Qualifier",
  "simpleDatatype": "Char",
  "core": "Perm",
  "_links": {
    "self": {
      "href": "/mdr/sdtmig/3-2/datasets/VS/variables/VSPOS",
      "title": "Vital Signs Position of Subject",
      "type": "SDTM Dataset Variable"
    },
    "codelist": [
      {
        "href": "/mdr/root/ct/sdtmct/codelists/C71148",
        "title": "Version-agnostic anchor resource for codelist C71148",
        "type": "Root Value Domain"
      }
    ],
    "modelClassVariable": {
      "href": "/mdr/sdtm/1-4/classes/Findings/variables/--POS",
      "title": "Position of Subject During Observation",
      "type": "Class Variable"
    },
    "parentProduct": {
```


Extract

VSPOS.json – a JSON file returned by an API request to the CDISC library containing standard (per SDTM-IG version 3.2) metadata for VSPOS

```
import json

# Use OPEN function to create FILE
object
file = open(vspos.json)

# Use LOAD function to convert to
dictionary object
jobj = json.load(file)

# Get C-code of codelist
href =
jobj['_links']['codelist'][0]['href']

code = href.split(sep='/')[-1]
```

Everything is an object - Datasets

- *Pandas* is a third-party module that supports data manipulation of one-dimensional and two-dimensional data structures through Series and DataFrame classes, respectively.
- Two-dimensional DataFrame object is like a SAS data set, but both columns and rows can have labels.
- Must first be downloaded and installed, then imported with the import statement in the program

```
import pandas as pd
```

- Like all other objects, DataFrame objects can be referenced with a variable!

Everything is an object - Datasets

Several ways to create a DataFrame object

- Constructors
  - List of dictionaries
    - `df = pd.DataFrame([{'a':1, 'b':2},{'a':5, 'b':10}])`
  - Dictionary of lists
    - `df = pd.DataFrame({'a':[1,5], 'b':[2,10]})`
- Read Excel workbook using Pandas' read_excel function
  - sdtmct=pd.read_excel('SDTM_Terminology_2018-06-29.xlsx', sheetname=1, usecols=[0,1,2,3,4,7], names=['Code','CodeListCode','Extensible','Name','CodeListItem','Decode'])
- Read CSV file using Pandas' read_csv function
- Read SAS file using Pandas' read_sas function

What can we do with data sets?

- Subset variables
  - `drop code codelistitem ;`
  - `sdtmct.drop(['code', 'codelistitem'], axis=1)`

  - `keep code codelistitem decode ;`
  - `sdtmct[['code', 'codelistitem', 'decode']]`
- Rename variables
  - `rename code = termcode codelistitem = term ;`
  - `sdtmct.rename(columns={'code': 'termcode', 'codelistitem': 'term'})`
- Sort
  - `proc sort data=ae ; by usubjid descending aeterm ;`
  - `ae.sort_values(by=['usubjid','aeterm'],ascending=[True, False])`

What can we do with data sets?

- Boolean filtering
  - `set suppae(where=(qnam eq 'qnam1'))`
  - `suppae[suppae['qnam'] == 'qnam1']`
- Remove duplicates
  - `proc sort nodupkey data=vs ; by usubjid vstestcd ;`
  - `vs.drop_duplicates(subset=['usubjid', 'vstestcd'])`
- Merge
  - `merge ae(in=inae) suppae(in=insupp) ; by usubjid; if inae ;`
  - `ae.merge(suppae,how='left',on='usubjid')`

What can we do with data sets?

- Transpose
  - `proc transpose data=suppdm out=suppdm2; by usubjid; id qnam; var qval;`
  - `suppdm2=suppdm.pivot(index='usubjid',columns='qnam', values='qval')`
- Summarize
  - `proc summary data=lb sum mean; var lbstresn;`
  - `lb.lbstresn.agg(['sum','mean'])`

What can we do with XML? Read!

severity.xml

```
<CodeList Name="Severity/Intensity Scale for …" DataType="text">
   <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>
```

The xml.etree.ElementTree module has two classes:

- ElementTree – The entire XML tree as a whole
- Element – An individual Element

```
import xml.etree.ElementTree as et

# parse function yields an ElementTree object
tree = et.parse('severity.xml')
```

getroot method returns an Element object from an ElementTree object
tag property returns a string object (name of the tag) from an Element object
attrib property returns dictionary from an Element object

```
rootelement=tree.getroot()
roottag=rootelement.tag ()
rootattrib=rootelement.attrib
```

```
CodeList
{'Name': 'Severity/Intensity Scale for…',
'DataType': 'text'}
```

Several methods for iterating through elements
text property returns the value of an element
get method returns the value of an attribute

```
for cli in rootelement.findall('CodeListItem):
   decode=cli.find('Decode')
   tran=decode.find('TranslatedText')
   text=tran.text
   alias=cli.find('Alias')
   aliasname=alias.get('Name')
```

```
Grade 1

C41338
```

What can we do with XML? Write!

severity.xml

```
<CodeList Name="Severity/Intensity Scale for …" DataType="text">
   <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>
```

SAS

```
if first.name then put '   <CodeList Name="' name +(-1)'"
DataType="' datatype +(-1) '">' ;
```

```
put '   <CodeListItem CodedValue="' codedvalue +(-1) '">' ;
put '    <Decode>' ;
put '     <TranslatedText>' decode '</TranslatedText>' ;
put '    </Decode>' ;
put '    <Alias Name=" code +(-1) '"/>' ;
put '   </CodeListItem>' ;
```

```
if last.name then do ;
  put '    <Alias Name="' codelistcode +(-1) '"/>
  put '   </CodeList>' ;
end ;
```

Python

```
import xml.etree.ElementTree as ET
CL = ET.Element('CodeList', Name=name, DataType=datatype)
```

```
CLI = ET.SubElement(CL,'CodeListItem',CodedValue=codedvalue)
DEC = ET.SubElement(CLI,'Decode')
TT  = ET.SubElement(DEC,'TranslatedText')
TT.text = decode
ALIASCLI = ET.SubElement(CLI,'Alias',Name=code)
```

```
ALIASCL = ET.SubElement(CL,'Alias',Name=codelistcode)

print (ET.tostring(CL,encoding='unicode'))
```

Conclusion

How do we learn a new object-oriented language like Python, coming from SAS?

- Object-oriented paradigm is fundamentally different from that of SAS
  - They do share basic programming constructs such as loops, conditions, etc.
  - Variable types may be the bridge from SAS to Python
  - Everything is an object, so what are its methods and properties?
  - User community contribution and documentation
- Approach
  - Get to know a language's core (e.g. syntax, built-in functions and types, etc.)
  - Research classes and modules of interest
- Documentation
  - docs.python.org
  - pandas.pydata.org/docs
  - lxml.de
  - Python-izing the SAS Programmer (PharmaSUG 2019)
  - Python-izing the SAS Programmer 2: Objects, Data Processing, and XML (PharmaSUG 2020)

Name: Mike Molter
Affiliation: Labcorp
Contact Number: 919-414-7736
E-mail: molter.mike@gmail.com
Twitter: False
LinkedIn: True