Using the SBCorpus Module in Python

by Kevin Schäfer

Getting started: Importing the SBCorpusReader

If you are using Python from a command line, navigate to the folder containing the SBCorpus files using *cd* (change directory) followed by the path to the folder containing this manual. Initiate a Python session by typing Python3 into the command line.

```
user$    cd C:\\path\to\SBCorpus\
user$    python3
```

Once a Python session has begun, you can import the SBCorpusReader by calling the following (You should be able to copy anything that appears in dotted lines in this manual by selecting the text only and pasting into the terminal – with CTRL + SHIFT + V on a Windows/Linux system. In IDLE or Anaconda, you should be able to paste code into the script window with CTRL + V and execute from there.):

```
>>>    from SBCorpus import SBCorpusReader as SBC
>>>    SBC = SBC()
```

The first line above imports the SBCorpusReader class from the file named SBCorpus. As a shorthand, we can import it under the name SBC to avoid having to type out the full name each time we invoke a method from this class. Following 'as', we could assign any variable name as long as we consistently replace 'SBC' in the following code with our alternative variable name. For instance, the method 'SBC.printSubset()' should be called with 'S.printSubset' if you choose to import the class as 'S' instead of 'SBC'.

The second line instantiates the class. We want SBC to refer to an instance of the class so that we can manipulate the corpus using the methods. Since we are no longer using the original value of SBC (the uninstantiated class), we can recycle this variable name and use it for the instantiated class. However, you may also choose to select a unique variable here (e.g.: '*S = SBC()*' to instantiate the class as 'S'). Be aware that the execution of the second line will take several seconds, since all the corpus files must be read into Python.

Basics

All of the following methods (that is, functions belonging to the class *SBCorpusReader*) belong to one of two types: "get-methods" and "print-methods".

Get-methods take arguments that return data objects extracted from the Santa Barbara Corpus. In order for these methods to be useful, they must be assigned to a variable. Calling/executing the function may print the information on the screen, depending on how you are using Python. However, in order for the data to be manipulated, you must assign the information to a variable:

```
>>>    LenoreIUs = SBC.getIUs(participant='NAME=LENORE')
>>>    teenTexts = SBC.getTexts(participant='AGE=11:19')
```

Above, 'LenoreIUs' is name of a variable containing all Ius spoken by corpus participants with the name 'LENORE'. The variable 'teenTexts' contains all corpus texts including teenage participants.

Print-methods take similar arguments, but display information that cannot be further manipulated. The object containing the output is a string presenting the information in a way designed to be human-readable. Thus, it is not useful to assign the output of print-methods to a variable, since these methods output to the screen, not to an information structure.

```
>>>     SBC.printParticipants('NAME=TOM')
```

Calling the above prints the output:

```
EDUCATION: B.A.
ID: 103
HOMESTATE: MA/NM
TEXTS: 32
YEARSEDUCATION: 16
NAME: TOM, TOM_1
CURRENTSTATE: NM
AGE: 60
HOMETOWN: Boston/New Mexico
ETHNICITY: WHITE
GENDER: M
OCCUPATION: Graphic Designer

EDUCATION: M.A.
ID: 104
HOMESTATE: NJ
TEXTS: 32
YEARSEDUCATION: 18
NAME: TOM, TOM_2
CURRENTSTATE: NM
AGE: 70
HOMETOWN: Irvington
ETHNICITY: WHITE
GENDER: M
OCCUPATION: Semi-Retired Consultant, Government

EDUCATION: LLB
ID: 105
HOMESTATE: SD
TEXTS: 32
YEARSEDUCATION: 22
NAME: TOM, TOM_3
CURRENTSTATE: NM
AGE: 68
HOMETOWN: Pine Ridge Reservation
ETHNICITY: NATIVE AMERICAN, WHITE
GENDER: M
OCCUPATION: Retired Judge, Legal Arbitrator, Hearing Officer
```

Exploring the Metadata:

SBCorpusReader.getParticipants(identifier, [info])

This method takes maximally an identifier string argument designating search terms and an output identifier restricting output information as optional input arguments. It returns a list of each participant in the corpus that meets the criterion/criteria specified – either with all the information or a specified target. If the method is called without parameters, all of the participants are returned.

The **syntax of the string** must be as follows:
The string must contain an ‘=’separating a search parameter on left from a value on right

Demographic parameters must be one of the following:

ID	unique numeric identifier for a participant (*0001-0213*)
NAME	participant's pseudonym (e.g. *LENORE*)
GENDER	gender of the participant (*M* or *F*)
AGE	participant's age in years (*11-101*)
HOMETOWN	participant's hometown (e.g. *SHREVEPORT*)
HOMESTATE	participant's home state, hometown location (e.g. *IL*)
CURRENTSTATE	the state where the participant currently lives
EDUCATION	the level of education that the participant has completed i.e.: BA, BS, College, Some College
YEARSEDUCATION	the number of years of education
OCCUPATION	occupation of the participant
ETHNICITY	race/ethnicity of the participant
TEXTS	numeric identifier for text featuring the participant (*1-60*)

Any numeric values can be indicated as a range with a colon 'AGE=12:24' includes participants ages 12 – 24, including those aged 12 and 24

To indicate that the argument is a string, the argument must be contained in single or double quotation marks within the parentheses:

```
>>>     SBC.getParticipants('AGE=51')
```

– or –

```
>>>     SBC.getParticipants("GENDER=F")
```

The second argument is optional. If no argument is provided for 'info', then all of the demographic information for all speakers meeting the search terms will be returned. The output will be a list (indicated by brackets) containing tuples (an immutable list indicated by parentheses). Each tuple represents a participant in the corpus, containing all the available demographic information for the participant.

If the 'info' argument is provided, then only that piece of information will be returned in a list. This argument must be a string (contained within single or double quotation marks). The optional argument must appear second in order, or alternatively, following 'info=' (not between quotation marks). Since the argument must be a string, the '=' must precede a term between quotation marks. Try the following examples:

```
>>>     no_bob = SBC.getParticipants("AGE=30:34,NAME!=BOB", 'ID')
```

(Returns a list of participant IDs for everyone in the corpus between the ages of 30.0-34.9[1], excluding any participants with the corpus pseudonym 'Bob', assigns to variable 'no_bob')

```
>>>     genders=SBC.getParticipants('GENDER=F,GENDER=M', info='GENDER')
```

1 All of the participants' ages are available as integers only.

(Returns a list of unique genders in the corpus; namely: ['F', 'M', ''']. Assigned to ‘genders’.)

SBCorpusReader.printParticipants(identifier)

This is the print-method equivalent. It takes parameters in the form of a string, as above, and returns all demographic information for all participants that meet the search criteria.

```
>>>     SBC.printParticipants('ID=5')
```

Calling the above prints the output:

```
EDUCATION: college
ID: 5
HOMESTATE: CA
TEXTS: 2, 31
YEARSEDUCATION: 16
NAME: JAMIE
CURRENTSTATE: CA
AGE: 30
HOMETOWN: Walnut Creek
ETHNICITY: WHITE
GENDER: F
OCCUPATION: dancer
```

Extracting parts of the corpus

Get-methods

Several methods are available to subset the corpus based on various criteria. These functions are similar in a number of ways. All of them begin with ‘get’, followed by the unit of interest: ‘Texts’, ‘Turns’, ‘IUs’, or ‘Words’. Names are case-sensitive; make sure to prefix the functions with whatever you have named the corpus in place of ‘SBCorpusReader’.

All of the methods can be called without arguments, returning the corpus in its entirety. As an exception, the ‘getWords’ method returns only IUs containing words, whereas the other three get-methods return the SBCorpus in its entirety. The arguments that can be passed with each method are generally similar, with unique arguments available only at relevant units of organization. Because these methods allow a large number of arguments, all arguments should be identified. For example, extracting all IUs containing overlap from a subset of the corpus that we have already extracted and assigned to a variable ‘txts1thru4’, we would identify the subset with ‘subset=txts1thru4’ (without parentheses) and the our search parameters with ‘containing='manner=OVERLAP'’ (with plain single/double quotation marks around the search parameters). For example:

```
>>>     overlap1thru4 = SBC.getIUs(subset=txts1thru4,
                                   containing='manner=OVERLAP')
```

All get-methods accept the following basic arguments:

argument		description
subset	-	*a subset of the corpus – the output of another get-method* *syntactically, either a variable to which a subset has been assigned* *or you can call a get-method inside of another get-method*
textlist	-	*a list of texts to narrow the search space* *syntactically, it can be a single integer, a list of integers, or a string* *e.g.: 4 is the same as '4' [2, 3, 4, 6] is the same as '2:4,6'*

participants	*a string indicating search parameters for participants* *syntactically, the string must be between single/double quotation marks* *for syntax within the string, see the 'indentifers' argument above in getParticipants*

Additionally, all arguments in get-methods narrow the scope of extraction. Passing an argument for participants and for texts limits the search to only the specified participants if they appear in the specified texts.

Units for extraction

SBCorpusReader.getTexts(subset, textlist, participants)

Optional input options: the basic get-method arguments (subset, textlist, participants)
Output: corpus including only texts meeting the criteria
e.g.:

```
>>>    lenore=SBC.getTexts(textlist='1:30', participants='NAME=LENORE')
```

The above creates a variable 'lenore': texts in the first half of the corpus including Lenore

```
>>>    allTexts = SBC.getTexts()
```

SBCorpusReader.getTurns(subset, textlist, participants, turnlist, IUlist, containing, afterTurn, beforeTurn, before, after, at, minlength, maxlength)

Optional input options, in addition to optional parameters *subset*, *textlist*, and *participants*:

arguments		description
turnlist	-	*quasi-turns to include in the output – a turn number/range for each text* *syntax: integer, list/tuple of integers, or string indicating a number/range*
IUlist	-	*quasi-turns to include based on the IUs they contain* *see 'getIUs' for details* ***note:** since the output unit of 'getTurns' is the quasi-turn, IUs outside of the specified range may appear in the output*
containing	-	*quasi-turns to include, identified by content of the words in the turn* *see 'getWords' for details* ***note:** since the output unit of 'getTurns' is the quasi-turn, turns in output include IUs and words not meeting search parameters*
afterTurn	-	*quasi-turns that the output turns **follow***
beforeTurn	-	*quasi-turns that the output turns **precede*** *syntax: 'getTurns' corpus subset or an embedded 'getTurns' call*
before	-	*timepoint which all quasi-turns in the output **precede***
after	-	*timepoint which all quasi-turns in the output **follow***
at	-	*timepoint which all quasi-turns in the output **contain***

minlength - *minimum length for quasi-turns in the output*
maxlength - *maximum length for quasi-turns in the output*
syntax: float or integer only

Output: subset of the corpus including only turns that meet the criteria specified by arguments

Some examples:

```
>>>  afterQ = SBC.getTurns(textlist=13,
                            afterTurn=SBC.getTurns(textlist=13, containing='dt=?'))
```

The above call first finds all turns containing a word '?' at the 'dt' level of representation in text SBC013. Then, it searches text SBC0013 for all turns following turns containing an IU with rising intonation. *Only turns following rising intonation are returned*.

Notice that the code includes the argument 'textlist=13' at two levels: once in the main call to the method, another time in the embedded call. This helps to narrow the search to the same domain, speeding up the processing time.

Another way to write the code above follows:

```
>>>  containsQ = SBC.getTurns(textlist=13, containing='dt=?')
>>>  afterQ = SBC.getTurns(textlist=13, afterTurn=afterQ)
```

One benefit to breaking up the complex, embedded call is that, with the code immediately above, we have two objects: all of the turns in SBC013 with rising-intonation in at least one IU ('containsQ') and another with all IUs that follow ('afterQ'). If we want a combined object later, we can use *SBCorpusReader.combineSubsets(containsQ, afterQ)*. See '*combineSubsets'*.

Another example:

```
>>>  at985_3 = SBC.getTurns(textlist=48, at=985.3)
>>>  SBC.printSubset(at985_3)
```

The code above prints the following in a human-legible format:

```
Mickey Mouse Watch (983.413 - 988.479)
983.413   987.281   JUDY;     ... You look good in it .
987.281   988.479             .. [Now |let] me take a picture .
```

It also saves the turn at which the timepoint 985.300 lies in text SBC048 in a format that can be more easily manipulated in Python.

SBCorpusReader.getIUs(subset, textlist, participants, turnlist, IUlist, containing, after, at, before, maxlength, minlength)

Optional input options, in addition to optional parameters *subset*, *textlist*, and *participants*:

arguments		description
turnlist	-	*quasi-turns that IUs in the output must belong to*
		see 'getTurns' for details
		note: *since the output unit of 'getIUs' is the IU, turns may appear in part*

IUlist - *IUs to include in the output – a turn number/range for each text*
syntax: integer, list/tuple of integers, or string indicating a number/range

containing - *IUs to include, identified by content of the words in the IU*
see 'getWords' for details
***note:** since the output unit of 'getIUs' is the IU, turns in output may include words not meeting the search parameters*

before - *timepoint which all IUs in the output **precede***
after - *timepoint which all IUs in the output **follow***
at - *timepoint which all IUs in the output **contain***
minlength - *minimum length for IUs in the output*
maxlength - *maximum length for IUs in the output*
syntax: float or integer only

Output: subset of the corpus including only IUs that meet the criteria specified by arguments

An example:

```
>>>  TOM_IUs = SBC.getIUs(participants='NAME=TOM', minlength=2)
```

The code above creates a variable called 'TOM_IUs' containing only IUs longer than 2.000 seconds spoken by corpus participants with the pseudonym 'Tom'.

SBCorpusReader.getWords(subset, textlist, participants, containing, tier, aslist, unit, fromstart, fromend)

Optional input options, in addition to optional parameters *subset*, *textlist*, and *participants*:

arguments - description
turnlist - *quasi-turns that words in the output must belong to.*
see 'getTurns' for details
***note:** since 'getWords' outputs words, turns may appear in part only*

IUlist - *quasi-turns to include based on the IUs they contain.*
see 'getIUs' for details
***note:** since the output unit of 'getTurns' is the quasi-turn, IUs outside of the specified range may appear in the output*

containing - *words to include in the output, identified by content or type*
syntax: string, preceded and followed by single/double quotation marks
**the string must contain '='.*
Left of the '=' is the search scope
Right of the '=' must be a word or a regular expression
Regular expressions must be indicated with 'r'' before the terminal and '' after the term within the larger string.
The argument string must use double quotation marks in this case.

See examples for more details

"dt	=	*r'regex' "*	
'dt	=	*string'*	Search the DT transcription of words
"word	=	*r'regex' "*	
'word	=	*string'*	Search the orthographic word
'manner	=	*string'*	Searches manner tiers
'POS	=	*string'*	Searches part of speech

aslist - *returns words as a list for various purposes.*
syntax: True or False, without quotation marks.
False returns a corpus subset readable by get-methods and print-methods
True returns a list useful for quantification

unit - *determines the organization of list output if 'aslist=True' is passed.*
syntax: string. 'IU' returns a list (complete output) of lists (IUs) of words
'word' returns a list (complete output) of words

tier - *determines which level of representation output words should have.*
syntax: string. 'dt' or 'word'

fromstart - *includes only the first x number of words per IU*
fromend - *includes only the last x number of words per IU*
syntax: integer
note: *as with all of the get-methods, arguments are subtractive*
Specifying fromstart and fromend will return only mid-IU words
from IUs with few enough words

Output: subset of the corpus including only words that meet the criteria specified by arguments

A few examples:

```
>>>  maleOverlap = SBC.getWords(participants='GENDER=M',
                                 containing='manner=OVERLAP')
```

The code above creates a variable called 'maleOverlap'. This contains a corpus subset of only words spoken by male participants overlapping another transcribed sound, audible action, or IU. 'maleOverlap' can still be printed with '*SBCorpusReader.printSubset*', but if we add the arguments 'aslist=True, unit='IU', tier='dt'', we get a list of IUs:

```
>>>  maleOverlap = SBC.getWords(participants='GENDER=M', aslist=True,
                        unit='IU', tier='dt', containing='manner=OVERLAP')
>>>  len(maleOverlap)
```

Using the builtin python 'len' function, we can count the number of IUs that men produce in the Santa Barbara Corpus in overlap. The code above returns that number: 31,852. To get the number of words in overlap, we would pass the argument 'unit='word'' to get 165,737.

After a subset object has been created, it can later be converted to a list with '*listWords*'.

Using regular expressions:

```
>>>    ha_words = SBC.getWords(textlist=23, aslist=True,
                                        containing="word=r'^ha',word!=have")
>>>    print(ha_words)
>>>    'have' in ha_words
```

The code above makes use of regular expressions and simple comparison of strings. In the argument '*containing="word=r'^ha',word!=have"*', there are two parameters. The term word=r'^ha' uses the regular expression '^ha' to search for words at the orthographic level of representation that begin with 'ha' (including a word represented in DT as 'h[aven't', for example). The second term removes from that subset of words all orthographic words 'have' (including 'ha@ve' but not 'haven't', for example). When we print the object, we see a list of all words beginning with 'ha' in SBC023, excluding the word 'have'. Indeed, the third line of the code above ('have' in ha_words) returns 'False', telling us that 'have' does not appear in the list.

Regular expressions can also be used at the 'dt' tier of representation to isolate words with certain characteristics. Regular expressions can be used to isolate words with a specified number of '@' symbols, representing a pulse of laughter, or to exclude words with overlap beginning before the word ("dt!=r'^\|'"), at the start of the word ("dt!=r'^\['"), or mid-word ("dt=r'[a-zA-Z]\['"). Matches can include or exclude search terms at various levels of representation, meaning that words can be extracted with certain phonetic properties and removed from the extraction based on part of speech. These terms need only be separated by a comma.

Manipulating extracted parts of the corpus

Creating new subsets based on other subsets

SBCorpusReader.combineSubsets(excerpt1, excerpt2)

Input: one or two corpus subsets, passed as separate arguments
or a list of corpus subsets, passed as a single arguments
subsets can be indicated by a variable name or by an embedded 'get-method'
Output: a combined subset created from the 2 or more input subsets

Some examples:

```
>>>    Marci = SBC.getTurns(participants='NAME=MARCI')
>>>    Ken = SBC.getTurns(participants='NAME=KEN')
>>>    Ken_after_Marci = SBC.getTurns(participants='NAME=KEN', afterTurn=Marci)
>>>    Marci_b4_Ken = SBC.getTurns(participants='NAME=MARCI', beforeTurn=Ken)
>>>    Marci_Ken = SBC.combineSubsets(Marci_b4_Ken, Ken_after_Marci)
>>>    SBC.printSubset(Marci_Ken)
```

In the code above, a subset of the corpus is built up from smaller parts. Two basic subsets are created: 'Marci', a subset of all the quasi-turns produced by Marci, and 'Ken', a subset of all of the quasi-turns produced by Ken. In the next two lines, we extract all the quasi-turns produced by Ken after a quasi-turn by Marci ('Ken_after_Marci') and the complement:

('Marci_b4_Ken'). Finally, these two subsets are fused together to create 'Marci_Ken', a subset of all the turn pairs in which the first turn is spoken by Marci, followed by a turn from Ken. When we print this subset, we see what appears to be a dialogue between Marci and Ken but what truly represents only select quasi-turns from text SBC013.

If subsets have overlapping elements, this is sorted out and the element appears only once in the output of 'combineSubsets'.

SBCorpusReader.getWindow(subset, castunit, outputunit, size, shift)

This method breaks a subset into small "windows" or double-paned timeslices
Note that this method make take a long time to process, especially if multiple texts are input

Optional input arguments (*marks default):

argument		description	syntax
castunit	-	*unit to measure window*	*string: 'word', 'IU'*, 'turn', 'second' or 's', 'millisecond' or 'ms', 'minute' or 'm'*
outputunit	-	*unit of windows in output*	*string: 'word', 'IU'*, 'turn'*
size	-	*number of units in each pane*	*integer or float (*10)*
shift	-	*number of units to shift window*	*integer or float (*5)*

The output is a list of subsets of the corpora, based on the arguments' time criteria

```
>>>  windows = SBC.getWindow(subset=SBC.getTexts(textlist=33), castunit='s',
                                                  outputunit='IU', size=15.000, shift=5)
>>>  words_per = []
>>>  for w in windows:
...      words_per.append( len( SBC.listWords( w, IUs=False)))
>>>  print(words_per)
```

The windows are not very useful unless we do something with them. In the code above, windows are generated for text SBC033 that are 30.000 seconds long, each window offset by 5 seconds (*shift*) from the preceding window. The *outputunit* is 'IU', meaning that some turns may be split by the window boundary. In the second line, we create an empty list 'words_per'. Using a for-loop in the next two lines, we append the number of words for each window. Finally, we can print our list (words_per) – an approximation of speech rate (words per half-minute).

```
>>>  windows = SBC.getWindow(subset=SBC.getTexts(textlist=33), castunit='word',
                                                  outputunit='word', size=18, shift=9)
>>>  percent_overlap = []
>>>  for win in windows:
...      SBC.printSubset(win)
...      ovlp = len( SBC.getWords(subset=win, aslist=True, unit='word',
                                                  containing='manner=OVERLAP'))
...      total = len( SBC.listWords(win, IUs=False))
...      percent_overlap.append( float(ovlp) / float(total))
>>>  print(percent_overlap)
```

The code above generates a window list, then uses a for-loop to print each window. Within each for-loop, only the overlapped words are extracted from each window. Each version of the window – overlapped words and all types of words – is converted into a list in one of two ways: the 'listWords' method or by specifying list output within 'getWords'. Each version of the list is counted with the builtin python 'len' function, then the proportion of overlapped words is calculated as a float. This float is appended to a list we created before the for-loop, which we can print at the end to see the proportion of overlap over time.

SBCorpusReader.listWords(words, tier, IUs)

This method takes as input a subset/window (*words*) and optional arguments *tier* and *IUs* The arguments (with *default options) are:

argument	options	description
tier	*'dt'*, 'word'*	determines the level of representation for the output
IUs	*True*, False*	determines whether the output is organized as an IU list

The output is a list of words, or a list of IUs, each IU represented as a list of words in that IU

This method can be used with the 'len' function to count the number of words or IUs in a given subset.

Viewing subsets and creating extracts from the corpus

SBCorpusReader.printSubset(subset, title, tier, timestamps, labels, numberedlines, decimal)

This method can be used to render a text or smaller extract in human-readable format.
The first argument (*subset*) is required, followed by a number of optional arguments (*default):

argument	options	description
title	*True*, False*	whether to include a title for each text in the subset
tier	*'dt'*, 'word'*	the level of representation for words
timestamps	*True*, False*	whether to display start and end times for each IU
labels	*True*, False*	whether to include participant labels
numberedlines	*True, False**	whether to include line numbers
decimal	integer (3*)	significant digits to display in time stamps

Output: a printed transcript that cannot be further manipulated.

Note: to copy an excerpt from terminal or command line, you will need to select the text and press CTRL + SHIFT + C to copy the text to clipboard.

Some examples and their output:

```
>>>  subset1 = SBC.getIUs(textlist=37, IUlist='30:35')
>>>  SBC.printSubset(subset1)
```

With default settings, we get the following output:

```
Very Good Tamales (50.561 - 61.327)
50.561    51.215    SHANE;      ... Yeah .
51.215    52.681                I:'m sure they're doing fine .
52.681    57.590    DOLORES;    ... [ |##] ## ~Shane .
56.493    56.745    KATE;       [#Oh:] .
57.590    59.113    DOLORES;    They can't stay put anywhere .
59.113    61.327    SHANE;      ... Why do you say that .
```

Tweaking some of the options, we can change the way the output looks:

```
>>>  SBC.printSubset(subset1, title=False, timestamps=False, labels=False,
                                                          tier='word')
```

With the arguments above, we get a version of the same text with no recognizable DT conventions:

```
 Yeah
I'm sure they're doing fine
   Shane
Oh
They can't stay put anywhere
 Why do you say that
```

We may wish to change the way timestamps look and add line numbers:

```
>>>  SBC.printSubset(subset1, tier='dt', decimal=0, numberedlines=True)
```

```
Very Good Tamales (50.6 - 61.3)
30    50.6    51.2    SHANE;      ... Yeah .
31    51.2    52.7                I:'m sure they're doing fine .
32    52.7    57.6    DOLORES;    ... [ |##] ## ~Shane .
33    56.5    56.7    KATE;       [#Oh:] .
34    57.6    59.1    DOLORES;    They can't stay put anywhere .
35    59.1    61.3    SHANE;      ... Why do you say that .
```