Binary Separated Value

B. Bean, N. Mezher, J. Summers & D. Toomey
University of Massachusetts Lowell — Software Engineering II
Prof. James Daly

April 2020

Motivation

The use of databases as a means for storing information has become ubiquitous in every field concerning data. One of the most common methods for analyzing sets of data from databases is to export it to a `.csv` (comma separated value) file format in order to manipulate the data via a spreadsheet program or language libraries. But despite all its draws, the `.csv` format has some significant drawbacks as well. The format is too bulky and inefficient for many applications, and it relies on a comma delimiter to separate data which can be problematic [1]. Our improvements upon the `.csv` format allow for users and programs to more efficiently store and utilize complex data. The end goal is to expedite communication between programs and disparate systems.

This document first outlines a new file format termed Binary Separated Value (BSVX), with the file extension `.bsvx`. This is not to be confused with the `.bsv` file format, which is a BASIC BSave Graphics file. The tailing `x` was chosen for convenience as it makes the name of our format, `.bsvx`, wholly unique. This format of data is delimited with byte markers which begin each field telling the library what kind of data is in the field and how long it is. We believed that through using byte markers, instead of plaintext character delimiters, a key issue with the `.csv` format—commas prematurely ending string fields—would be solved. The Binary Separated Value format is processed through a proprietary Python library called BSVXPY.

BSVX File Format Specification

Each `.bsvx` file contains a series of rows of headers or records. Each row begins with a byte marker denoting the type (i.e. header or record) and the number of fields within that row. Following the first byte marker of a row is a series of fields, each made up of two parts: a byte marker denoting the type and size of the data stored within the field, and the data itself. Some initial markers indicate that the size of the data is given in subsequent bytes. Once the length n is determined, those n bytes can be interpreted to match the field byte marker. Each row does not have to be the same length, the data can be jagged and parsers read as much data as is denoted by the first byte marker of each row.

At any time, the parser knows how many bytes it needs to read. There is never an instance where the parser needs to read bytes until it sees a particular character (as opposed to `.csv` or `.tsv` parsers, which look for commas or tabs respectively). Strings need neither end marks nor escape characters, and are stored in the UTF-8 format. The byte marker for strings denotes the number of bytes read, not the number of characters of the string itself. All numbers are stored in little endian order.

An abstract example of a `.bsvx` file row (header or record) looks like this:

blank	The quick brown fox... dog str	The quick brown fox... dog str	0 int	16 int	3.141900062561035 float

Table 1: An example of a `.bsvx` file row.

The same example of a `.bsvx` file row (header or record) but represented in hexadecimal:

0x00	0x2C	0x5468...6f67	0x88	0x90	0x20	0x0994	0x99	0x404914E4
blank	str	...val	short int	long int	...encoding	float	...value	float dec

Table 2: An example of a `.bsvx` file row in hex values.

The following table displays the implementation for each type of supported data in its own class. Bit ranges for each field are also provided; they are denoted by values ranging from 0 to 255. The first column gives the parser crucial context: what type of data follows the byte marker, and further, which *variant* on that type it is. For example, the short integer type is represented by numbers in the range 136-143. A 2 byte short integer is indicated by 138, 139 indicates a 3 byte integer, 140 indicates a 4 byte integer, etc. The second column illustrates how the range of values for a given type is affected by the magnitude of its offset. I.e. for a short integer, the second column entry is 136 + [0, 7]. The third column establishes the types of data that are supported, and the fourth column provides a brief description of each.

Range	**Form**	**Name**	**Description**
0		Blank	Possible implementation: NULL or 'empty string'
1-127	1-127	Short str	UTF-8 Encoded string of byte length 1-127
128-135	128 + [0,7]	Long str	1-8 bytes giving the length of a str, followed by said str
136-143	136 + [0,7]	Short int	An integer in the range of 0-7 bytes
144-151	144 + [0,7]	Long int	A zig-zag encoded integer using 1-8 bytes
152-159	152 + [0,7]	Float	IEEE-754 format float: 0 = half precision, 1 = single, 2 = double, 3 = triple
160-167	160 + [0,7]	Blob	1-8 bytes giving the length of binary data in bytes, followed by said data
168-183	168 + [0,15]	Header	Beginning of header with 0-15 fields
184-191	184 + [0,7]	Long header	1-8 bytes giving the number of fields in the header
192-207	192 + [0,15]	Record	Beginning of record with 0-15 fields
208-215	208 + [0,7]	Long record	1-8 bytes giving the number of fields in the record
216-255		Reserved	For future use

Table 3: Data types supported by the BSVX file format specification.

Implementation

As a generality, our Python library is similar to the `.csv` Python library. A writer function is passed a series of fields representing a header row. Subsequent binary values are decoded based on the corresponding type casts provided by the header. The library then extracts each of the fields from the dictionary object and outputs them in sequential order.

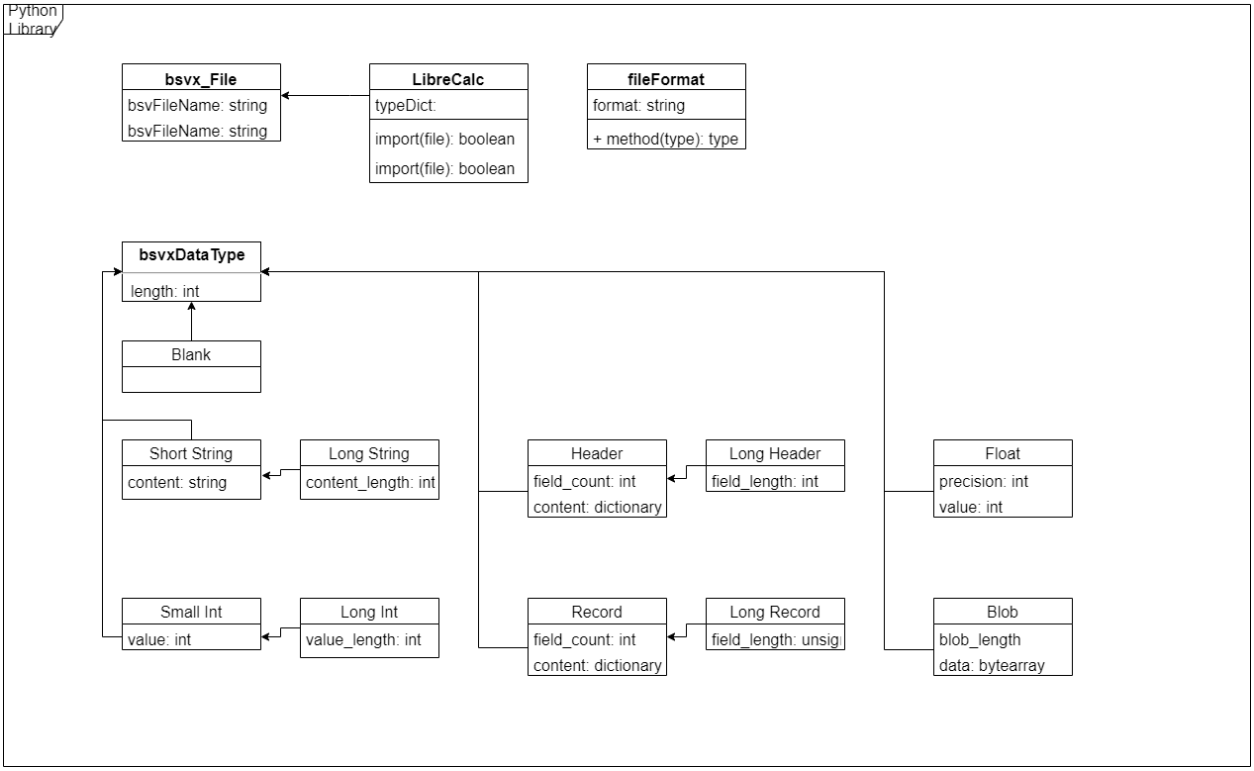

Figure 1: UML diagram outlining the classes and statstructures used in BSVXPY

A novel problem with a strongly typed encoding is the inability to handle undefined data types such as graphs or algorithms that are often used in spreadsheet applications. To solve this, the `.bsvx` file format manages unknown data types with a catch-all data type—Blob—which acts as a polymorphic object. These Blobs store raw binary data imported from spreadsheets or `.csv` files. By relying on the raw binary for unknown data types, BSVXPY can accommodate proprietary encodings associated with any third party application. As a note, this introduces the risk of `.bsvx` files being limited to one spreadsheet application when using Blob data types. This is because third party applications may use internal encodings which are unknown to other programs.

The current implementation of BSVXPY includes interface classes to hold data, as well as providing the user with the ability to read and write using files. The `Reader` class is designed to take in any `.bsvx` file and create a map of BSVXPY classes for the user to operate with. This is done by reading in the hexadecimal (hex) text of a `.bsvx` file and iterating through the data, creating classes as it traverses. When the file is first opened, the class reads in the first two hex characters on the first line to know how many fields to read in. A loop is then called to iterate over `i` columns of data. Each record contains hex values telling the `Reader` what class it is. `Reader` then creates a `bsvxDataType` class for the corresponding type by passing in the hex representation of the data to the class' constructor. Once the class is instantiated, it is appended to a map containing the string representation of the data as well as the class itself.

`Writer` is a class designed to write any `bsvxDataType` object to a file. Its implementation is simple, and is mostly handled by the `bsvxDataType` subclasses. Each subclass contains its own unique `write()` function, which is implemented differently depending on how the data is formatted. Each class you pass to `Writer`'s `writerow()` function will be written to the file in that order.

Development

In developing the BSVXPY Python library, the BSVX team took a number of precautions. At the highest level, the project was split into two different repositories on GitHub, so that BSVX's components and

documentation were entirely separate. The BSVXPY repository holds the BSVXPY Python library. And the DOCS repository contains efforts related to this document and generating the results shown later in this document.

To further isolate BSVX, the development of the BSVXPY Python library required the use of a Python virtual environment. This ensured that the end user did not install the incomplete module on their default Python environment. Specifics regarding the initialization and launch of a Python virtual environment, within the context of this project, can be found in the README portion of the BSVXPY repository.

Ensuring correctness, to the best of the BSVX developers, Travis-CI was utilized. Travis-CI allows for integration with GitHub repositories, and helped to automatically test BSVX's components. As for the tests themselves, the PYTEST package was utilized. The PYTEST package allowed the BSVX developers to easily and quickly construct unit and systems tests for the BSVXPY Python library. Different testing files were associated for each data type supported by the BSVX file format specification so that each component's functionality could be tested separated if needed. If all tests that were tested—typically all tests created—passed, Travis-CI would assign the repository a passing mark. Failing Travis-CI marks enabled BSVX developers to retroactively fix problems with some context as to where they happened. This continuous integration process made development safe and incremental.

Example `.bsvx` files were created to accurately account for testing basic functionality as well as edge cases. Initial builds emphasized basic functionality, such as correctly reading and writing basic data types. Blob types were handled after initial testing was completed. In every step of the process, the BSVX developers attempted to optimize memory and computational efficiency to improve performance outcomes.

Results and Discussion

We anticipated a user with minimal data file experience being able to perform `.bsvx` file conversions without loss of data or corruption of its ordering. While the later part of the above statement proved true, users must actually have sufficient familiarity with `.csv` or other data file types to integrate BSVXPY effectively. This is because the BSVXPY module is partially imcomplete and requires some manual data manipulation for writing to `.bsvx` files. The format does however maintain a capacity for the full integrity of its input data.

Dealing with undiscovered ambiguities in the format specification was a problem bigger than we initially antipated. These ambiguities were dealt with through tightening the specification and updating reference implementation. Parsing `.csv` files for conversion to `.bsvx` files, and vice-versa, involved numerous pitfalls. While there was an agreed upon standard format for `.csv` files, it didn't come about until 2005 and many `.csv` files still did not conform to it strictly. This complicated our attempt to ensure integrity and continuity between conversions for *all* `.csv` and `.bsvx` files. For instance, when converting a `.bsvx` file consisting of several strings of comma characters, our library had to ensure that none of the commas ended up being misinterpreted as delimiters. Properly following the specification ensured consistency and prevented this from happening, but rigorous testing with a myriad of files was necessary.

Another challenge fundamental to the `.bsvx` format was deciding how to handle Blob objects. It is not always clear what type of data a Blob should be deserialized as. We took the stance that the user should have context of the Blob's contents, so anything outside of the confines of the BSVX specification is not BSVX's responsibility.

An aspect we believed integral to BSVX was its compression of information. This was to be attained through storing fields in hexadecimal. To diagnose the successfulness of BSVXPY, we first decided to check if this goal was met. We started by creating `.csv` files of increasing sizes. Files predominantly filled with strings were chosen for comparison as the team knew that the BSVX file format specification left the most room for string types. Matching `.bsvx` files were generated for file size comparison.

What the team found surprised them. It turned out their initial hypothesis that the BSVX file format specification would save space, in comparison to the CSV file format, was incorrect. It was discovered that implementation resulted, on average, in `.bsvx` files twice as large as their `.csv` counter parts.

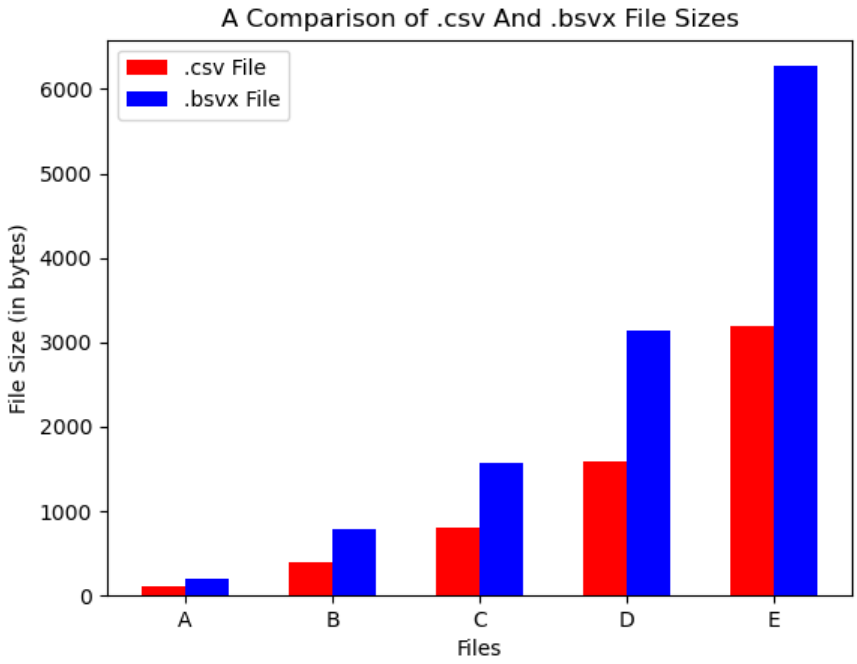

Figure 2: A graphic comparing files sizes between equivalent `.csv` and `.bsvx` files.

Worth mentioning again, is that these results represented a worst case scenario, as the BSVX file format specification anticipates strings of significant length.

But while BSVX failed in the fize size regard, it succeeded in solving an error fundamental to `.csv` files. BSVX sought to solve the limitations of the comma delimiter and did. Similarly, `.csv` and `.bsvx` files were manually generated, and consisting mostly of strings. Displayed below is a graph of expected versus actual fields for equivalent `.csv` and `.bsvx` files. `.bsvx` files match the expected number of fields, whereas `.csv` files decidedly do not. In fact, `.csv` files develop an increasingly larger margin of error with larger file sizes.

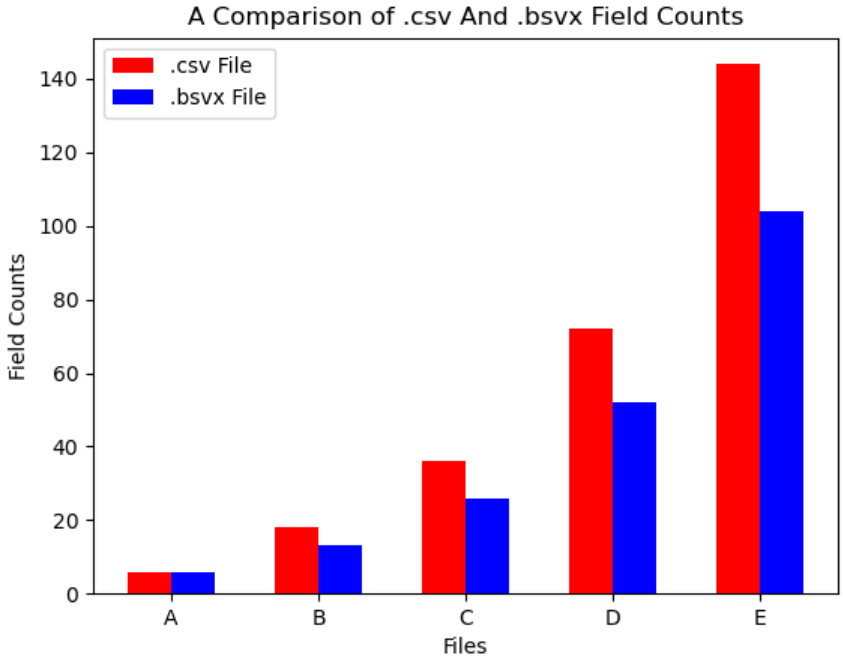

Figure 3: A graphic comparing field counts between equivalent `.csv` and `.bsvx` files.

File A represents a file with zero expected comma delimiter issues and as can be seen, both `.csv` and `.bsvx` files have an equal number of fields. But with files with a larger number of commas, the issue with the CSV standard becomes more pronounced. These errors could represent substantial problems in production environments. The BSVX file format specification, although more weighty, does its expected job of resolving comma delimiter issues.

Bash scripts were created to generate these key figures. These scripts take advantage of Python's comprehensive mathematics library NUMPY to aid with data collection. Figure generation is handled through the MATPLOTLIB Python library. The master script for figure generation script can be found in the scripts folder of our `docs` repository along with usage instructions in the form of a README. We offer this means of figure self-generation such that it is clear and self-provable that all figures in this document are fair.

Originally the BSVX team had planned on producing two deliverables. One was a new Python library able to read and write `.bsvx` files, according to the BSVX file format specification, and with the same functionality set as the Python CSV library. The second was a LibreOffice Calc extension, able to read `.bsvx` file data to and from LibreOffice Calc spreadsheets.

However, the BSVX team—like most everyone else—did not foresee the COVID-19 pandemic. With the pandemic's timing, the team was forced to change their implementation plans. The team decided to not pursue development of the BSVX LibreOffice Calc Extension. They believed it best to both limit the scope of the project to maintain a standard of code, and to diverge resources towards the BSVXPY Python library.

All of this was said to provide some context to the development of the BSVX LibreOffice Calc Extension. While not entirely necessary, the BSVX team hoped these remarks clarified the upcoming section's size. What follows is an outline for the implementation of a LibreOffice Calc extension for BSVX. Such resources remain freely open for development.

Future Work

Further iterations of BSVXPY would emphasize compactness. As it stands, the specification provides ample room for a number of data types. The specification need be refined if it is to have less of a footprint. Strings are one area where this issue is glaringly obvious.

One drawback of implementation through a Python library is the inability to easily parse and edit `.bsvx` files through a text editor. However, this issue is remedied through our proposed BSVX LibreOffice Calc Extension. LibreOffice is a free to use, open-source file editing platform similar to Microsoft Office. LibreOffice Calc is a program provided in the LibreOffice suite, and provides similar functionality to Microsoft's Excel program [4]. The BSVX LibreOffice Calc Extension would give LibreOffice Calc users the ability to read data from `.bsvx` files and export their spreadsheets to `.bsvx` files. Users would also have the ability to import, then convert `.csv` files into `.bsvx` files through the BSVX LibreOffice Calc Extension. The BSVX LibreOffice Calc Extension should be capable of ultimately converting between `.csv` and `.bsvx` files without loss or adulteration of information.

To illustrate the top-most point for user interaction with the BSVX LibreOffice Calc Extension, there is included a series of figures below. Figure 4 displays the default toolbar packaged with LibreOffice Calc. Figure 5 contrasts the differences between the default toolbar and a toolbar with the BSVX LibreOffice Calc Extension enabled. It can be seen that only two features are to-be added, in the form of two buttons. The proposed left button allows for importing, reading from, a `.bsvx` file and the proposed right button for exporting, saving to, a `.bsvx` file. Finally, Figure 6 provides a glance as to how the toolbar would look with the BSVX LibreOffice Calc Extension enabled.

The LibreOffice Calc project allows developers to create extensions using Python, which would let us extend LibreOffice Calc's functionality to include `.bsvx` file format support using our BSVXPY Python library. We would also use the UNO Python library, as it is necessary for any LibreOffice Calc extension development.

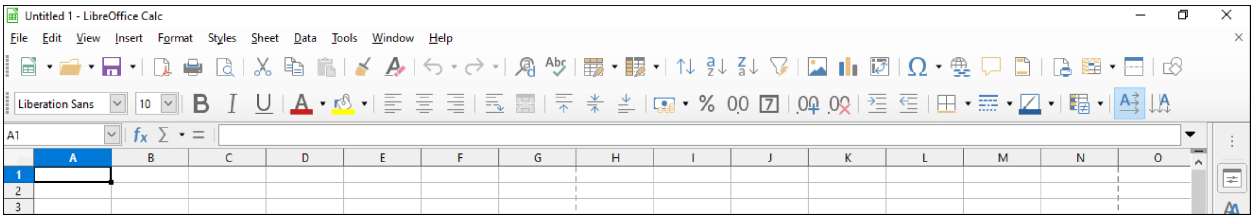

Figure 4: LibreOffice Calc's toolbar.

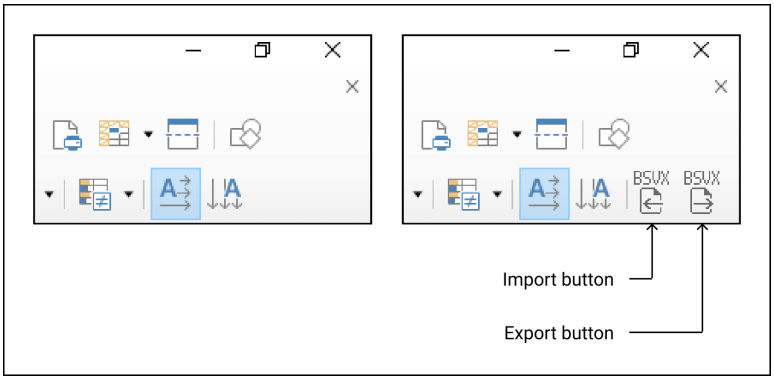

Figure 5: The BSVX LibreOffice Calc Extension would provide two additional buttons for importing and exporting.

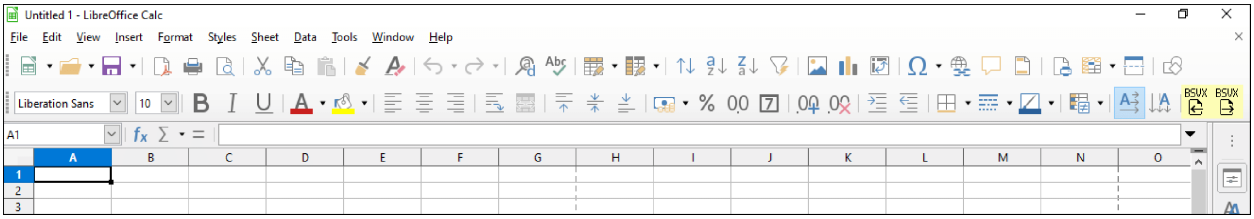

Figure 6: A mockup of LibreOffice Calc's toolbar with the BSVX LibreOffice Calc Extension enabled.

To export data to a `.bsvx` file, the BSVX LibreOffice Calc Extension would call functions from the UNO Python library to read cell data from LibreOffice Calc. The BSVXPY Python library would then be used to convert that data into binary separated values. Once the data is appropriately converted, it would be written to a file with the extension `.bsvx` and as named by the user. Figure 7 depicts the data flow for the exporting feature.

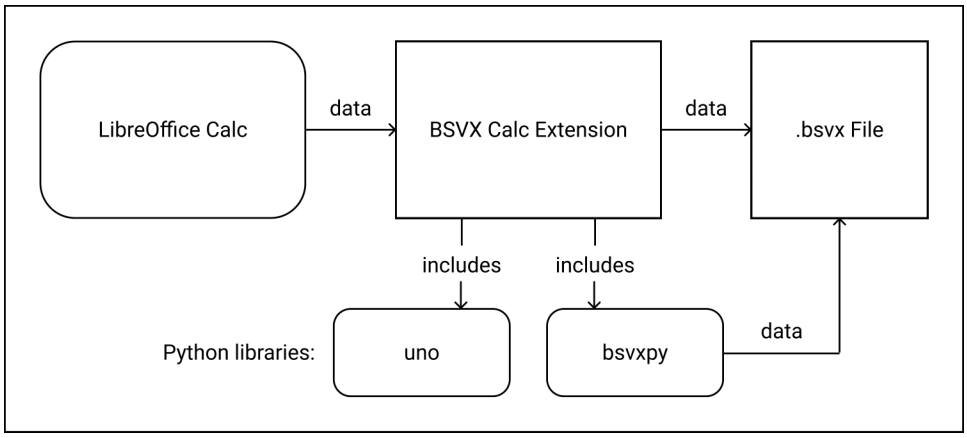

Figure 7: Scope of API and library calls for the BSVX LibreOffice Calc Extension in exporting data to a `.bsvx` file.

The importing feature would work the same way, but in reverse. The user would select a `.bsvx` file to import data from, and the BSVX LibreOffice Calc Extension would read data from that file, using BSVXPY and UNO to translate it from binary separated value data to cell data that LibreOffice Calc can read. The data flow for the importing feature is represented by Figure 8.

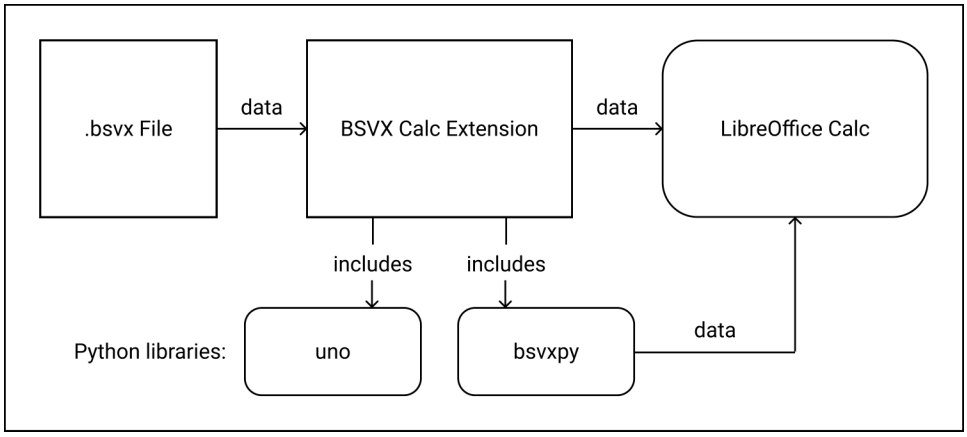

Figure 8: Scope of API and library calls for the BSVX LibreOffice Calc Extension in importing data from a `.bsvx` file.

This functionality would also allow the user to convert from `.csv` to `.bsvx`. If the user chose to import a `.csv` file into LibreOffice Calc, and export that file to a `.bsvx` file, that same data would become available in both files. Likewise, if the user chose to import a `.bsvx` file and export that file to a `.csv` file, that same data would become available. These changes would not significantly affect the overall architecture of LibreOffice Calc. The same base functionality of LibreOffice Calc would still be provided to the user once this extension was installed, just with the addition of importing and exporting features for the `.bsvx` format.

As a last note and concerning the prior discussed blob types, LibreOffice Calc may contain methods to interpret unknown fields upon reading the file, but more research on this is necessary.

Glossary

.bsvx The file extension associated with the Binary Separated Value file format.

.csv Comma-separated values file format often used for databases and spreadsheets.

.tsv Tab-separated values file format used for databases and spreadsheets.

Binary Separated Value The file format associated with the encoding protocols laid out in this document.

Comma delimiter Practice of using the ‘,’ character as a field separator to differentiate records in a file. Instances of a comma are always interpreted as a delimiter unless they appear in doubles quotes e.g. “1,0”.

Deserialization The process of decoding a `.bsvx` byte stream into its original data.

LibreOffice Calc An open-source application for manipulating spreadsheets. Developed and maintained by The Document Foundation.

Serialization The process of encoding data into a `.bsvx` byte stream.

References

[1] Coleman, Larry. "Why do we keep using CSV?" Software Engineering Stack Exchange, 14 Feb. 2011. https://softwareengineering.stackexchange.com/questions/47838/why-do-we-keep-using-csv

[2] "CSV File Reading and Writing." Python Docs, 12 Feb. 2020. https://docs.python.org/3/library/csv.html

[3] Daly, James and Meiners, Chad. "Binary Separated Value." UMass Lowell, 30 Jan. 2020.

[4] Guthrie, Gordon. "How to Work With LibreOffice Calc." TechRadar, 23 July 2012. https://www.techradar.com/news/world-of-tech/roundup/how-to-work-with-libreoffice-calc-1089870