# pySBOL3: SBOL3 for Python Programmers

Tom Mitchell,\* Jacob Beal, and Bryan Bartley

Raytheon BBN Technologies, Cambridge, MA 02138, United States

E-mail: tom.mitchell@raytheon.com

#### Abstract

The Synthetic Biology Open Language version 3 (SBOL3) provides a data model for representation of synthetic biology information across multiple scales and throughout the design-build-test-learn workflow. To support practical use of this data model, we have developed pySBOL3, a Python library that allows programmers to create and edit SBOL3 documents. Here we describe this library and key engineering decisions in its design. The resulting implementation is a compact and maintainable core that provides both a familiar, pythonic interface for manipulating SBOL3 objects as well as mechanisms for building additional extensions and representations on this base.

# Keywords

synthetic biology, standards, ontology, design automation, Synthetic Biology Open Language, Python

# Introduction

Synthetic biologists have a growing need to create machine-readable descriptions that interconnect information about engineered biological systems across multiple scales of resolution, e.g., from individual sequence features to functional units, thence to genetic regulatory networks, transformed strains, experimental conditions, and ultimately the plans of entire experiments and experiment campaigns. Likewise, there is an increasing need for traceable linkages between the various stages of a typical design-build-test-learn engineering workflow, e.g., tracing from model parameters to data analysis, thence to raw experiment data, protocol execution records, aliquots of organisms and reagents, and experiment plans and genetic designs. The Synthetic Biology Open Language version 3 (SBOL3) provides a standard representation for recording and sharing such information.<sup>1,2</sup> Effective usage of this standard, however, requires supporting software libraries that allow programmers to readily create and manipulate SBOL3 documents.

To this end, we have developed pySBOL3, a Python library implementing the SBOL3 data standard. The pySBOL3 library has been engineered to provide all core functionality for SBOL3 through a compact and maintainable implementation with a familiar, pythonic interface for manipulating SBOL3 objects, along with mechanisms for building additional extensions and representations on this base. The pySBOL3 library has now achieved a stable release status and is being used in a variety of synthetic biology software projects.

# Results

We first present the key design decisions in the implementation of pySBOL3, then describe how pySBOL3 represents SBOL data and handles extensions, followed by a discussion of RDF formats and data interchange.

#### Design of pySBOL3

The design of pySBOL3 is driven by a number of goals based on experiences in building and maintaining Python libraries for the SBOL2<sup>3</sup> standard: pySBOL/libSBOL<sup>4</sup> and pySBOL2. With each subsequent generation of the library, we have taken the opportunity to apply lessons learned and refine successful strategies and principles. The resultant design goals are outlined here and detailed below:

- Objects, not RDF: Provide an object-oriented programming interface that represents SBOL3 constructs as real Python objects, not as a veneer over an RDF graph, thereby allowing programmers to work intuitively with the objects and apply the full capabilities of the language.
- Simplicity: Keep the library focused on representation of SBOL3 constructs, avoiding the potential complexity and maintenance burden of adding "higher-level" helper functions, utilities, or other tooling. Such helpers and utilities are maintained in a separate project called SBOL-utilities, discussed later.
- True to heritage: Retain the core data representation techniques used in libSBOL, pySBOL, and pySBOL2, thereby simplifying the design of "bilingual" applications that need to work with both SBOL2 and SBOL3 and easing user and tool migration from SBOL2 to SBOL3.

#### **Objects**, not **RDF**

The SBOL3 standard is based on the Resource Description Framework (RDF),<sup>5</sup> which organizes data into collections of "triples", elemental statements of fact composed of a subject, predicate, and object. SBOL3 documents are RDF triple collections. Some SBOL libraries, like libSBOLj3,<sup>6</sup> interact with RDF by putting an object-like veneer over it to make the RDF appear as though it had been instantiated into objects. Data manipulations on these "objects" update the underlying RDF graph. This approach helps with handling non-SBOL triples intermingled with SBOL triples and makes serialization and deserialization very easy. On the other hand, this approach means the RDF graph must be queried repeatedly when accessing object attributes and generally means that some aspects of language functionality cannot be applied to the "objects."

In implementing pySBOL3, we have instead chosen to represent all SBOL3 objects as

Python objects. When an SBOL3 document is loaded into pySBOL3, the RDF contained in that document is translated into instances of Python classes that conform to the class definitions in the SBOL3 data model. The objects in a document are then converted and serialized to RDF when written to a file.

Likewise, pySBOL3 pays special attention to making the properties of SBOL3 objects behave as a Python programmer would expect. In particular, pySBOL3 allows the links between objects to be viewed as both a pointer to a Python object and an RDF URI. PySBOL3 also ensures that multi-valued RDF properties of an object can be manipulated in exactly the same way as a Python list, where appropriate. Together these design decisions allow the patterns for working with SBOL3 objects to be instantly familiar and intuitive for experienced Python programmers, as illustrated in Figure 1 and Figure 2.



Figure 1: Example of object manipulation in pySBOL3, showing reading a document, then using standard Python object idioms to search through and extract information from SBOL objects.

The tradeoff for these design decisions is that pySBOL3 incurs additional overhead when reading and writing files, which would not be the case if we used the object-like veneer approach. We judge this to be a desirable tradeoff, as it favors applications that need to perform complex in-memory creation and manipulation of SBOL objects, for which a library



Figure 2: Example of creating an SBOL object and setting its properties in pySBOL3 to represent a GFP expression cassette.

like pySBOL3 is critically necessary. At the same time, applications needing extremely rapid read and write speed still have the option of falling back to using the RDF libraries to work with SBOL on the triple level.

#### Simplicity

Our goal from the outset was to keep pySBOL3 focused on representing SBOL3 constructs. The prior libSBOL and pySBOL (a SWIG wrapper on libSBOL) had many convenience functions and macro-like functions. The pySBOL2 library was a port of pySBOL into native Python, and thus retained similar functions. This additional functionality, however, ended up committing the library to particular workflows and models of use for SBOL, decreasing its flexibility and increasing maintenance burden.

Convenience and macro-like functions do have a place in any library, however, so we chose early on to put those functions in a separate library, SBOL-utilities,<sup>7</sup> so that programmers could choose either to use them or to use only the core library. The SBOL-utilities library thus provides a layer on top of pySBOL3 that aggregates such functions, as well as a number of other supporting tools and utilities.

The separation between pySBOL3 and SBOL-utilities has allowed us to be more exper-

imental in our approach to utility functions. A number of contributors to SBOL-utilities have added a variety of useful tools that layer on top of the core pySBOL3 library. At the same time, there are other tools that use pySBOL3 without SBOL-utilities, making use of its functionality in other ways.

#### True to Heritage

Finally, in any matter where pySBOL3 is not constrained by the above design constraints, we have chosen to make pySBOL3 as similar as possible to the preceding libraries, libSBOL, pySBOL, and pySBOL2. In this way, pySBOL3 draws on the data representation utilized in libSBOL/pySBOL and then carried forward into pySBOL2, making it simpler to migrate between libraries or to work with both at once. For example, properties specified in SBOL3 that have cardinality greater than 1 are manipulated in pySBOL3 as a Python list (and not, for example, a set) and are generally named with a plural (even though the RDF property is singular).

#### Data Representation

The SBOL data model declares properties for all SBOL classes. Each of these properties has a type and a cardinality. For example, Component.features has type URI and cardinality ZERO OR MORE while Range.start has type integer and cardinality EXACTLY ONE. pySBOL3 represents SBOL properties as custom attributes that incorporate type and cardinality. These attributes act like native Python types (int, str, list).

Since SBOL3 is a strongly-typed language, while Python itself is dynamically-typed, pySBOL3 includes additional machinery for type-checking and data-validation that helps the user create valid SBOL data. The library provides this extra-layer of data validation by overriding the default behavior of Python attributes using specialized **Property** classes that handle type-checks, cardinality checks, and other validation functions. Additionally, these **Property** instances store their values in one of two dictionaries depending on its type. These custom attributes know how to dereference the dictionaries to either "get" or "set" their value, including type and cardinality checks. This dereferencing is transparent to the programmer, such that these attribute instances appear as native Python types with appropriate semantics based on that type.

Behind the scenes, the SBOL properties, and in fact all RDF properties whether they are SBOL properties or not, are stored in either a "properties" dictionary or an "owned objects" dictionary. If an SBOL property represents "composite aggregation" then the property is stored in "owned objects". All other SBOL properties and all RDF properties are stored in the "properties" dictionary (see Section 4 of the SBOL3 specification<sup>2</sup> for more details on property types and composite aggregation).

#### **SBOL** Extensions and Annotations

Some applications may have additional data requirements that are not fully covered by the SBOL data model. For example, a visualization tool that renders schematic depictions of genetic circuits may have additional layout and color parameters that need to be appended to the core SBOL representation of the circuits. To support such needs, pySBOL3 allows users to define extensions and annotations that extend the SBOL3 object classes in the library. Extensions are custom classes that extend the properties of a core class (i.e., the new class is a child of the SBOL3 Identified class), and annotations are custom properties that can be added to SBOL3 classes. This custom data can then be included, manipulated, and exchanged with other applications through SBOL files.

Programmers can write their own extensions and annotations using pySBOL3 by creating new classes and by adding additional properties to these classes. If a programmer defines their SBOL extensions and annotations in this way, then they get the benefit of type and cardinality checks provided by pySBOL3, and can manipulate their custom materials in the same way as SBOL3 classes. However, even if an application is unaware of these extension definitions, pySBOL3 will still conserve the extension data during file input/output operations and allow users limited access to the extension or annotation data. In this case, no type or cardinality checks can be performed because the extension definitions are unavailable. Regardless of whether the programmer defines an extension or annotation, however, the data is properly read from and written to RDF. To this end, pySBOL3 stores unknown RDF properties on SBOL objects in the hidden "properties" dictionary. Undefined annotations are not accessible through a published API, but they can be made accessible by defining custom properties, which then allow this RDF to be translated into the Python object model just like core SBOL3 properties and objects.

In addition to SBOL extensions, pySBOL3 will also successfully round trip (i.e., read and write identically) RDF triples in an SBOL file that are neither SBOL extensions nor SBOL annotations. There is currently no public API in pySBOL3 for accessing or modifying these non-extension, non-annotation triples, because they cannot be related to an SBOL3 object. They are stored in a hidden data structure and can be accessed by transforming the SBOL document into RDF.

#### **RDF** Formats and Interchange

Finally, the pySBOL3 library supports a variety of RDF formats to allow serialization and interchange with as many tools as possible. The library can read and write the RDF/XML, N-Triples, JSON-LD, and Turtle formats for RDF serialization, which the SBOL3 standards recommends as a minimum set for libraries. In addition, pySBOL3 also supports a sorted N-Triples format that is guaranteed stable, for better compatibility with version control systems. This allows easy interchange with other SBOL3 libraries and other RDF tools. Using these formats, cross-library data interchange has been verified with the Java SBOL3 library libSBOLj3 via the SBOL Test Suite.<sup>8</sup>

# Discussion

The pySBOL3 library provides a compact, pythonic interface for use of the SBOL3 standard. Development has been guided by ongoing interactions with developers making use of the library in a number of projects, reporting bugs and requesting features. As of this writing, the alpha and beta releases of pySBOL3 have been in use for over a year, and GitHub currently identifies 18 projects as directly dependent on pySBOL3, including Intent Parser,<sup>9</sup> LOICA,<sup>10</sup> Excel-to-SBOL,<sup>11</sup> and PAML.<sup>12</sup> Future development of the library will continue to be driven by user needs and the ongoing evolution of the SBOL3 standard.

# Author Contributions

Conceptualization and Methodology: T.M., J.B., B.B. Software: T.M. Validation: T.M., J.B., B.B. Writing: Original Draft: T.M., J.B., Review & Editing: T.M., J.B., B.B.

# **Conflict of Interest Statement**

Conflicts of Interest: None.

### Acknowledgement

This work was supported by Air Force Research Laboratory (AFRL) and DARPA contracts FA8750-17-C-0184, FA8750-17-C-0231, and HR001117C0095. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations. Views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Many users provided useful feedback and helped to shape the pySBOL3 package as it was being developed. A debt of gratitude is due to the developers of pySBOL and pySBOL2 for forging the path for pySBOL3.

# References

- McLaughlin, J. A.; Beal, J.; Mısırlı, G.; Grünberg, R.; Bartley, B. A.; Scott-Brown, J.; Vaidyanathan, P.; Fontanarrosa, P.; Oberortner, E.; Wipat, A., et al. The synthetic biology open language (SBOL) version 3: simplified data exchange for bioengineering. *Frontiers in Bioengineering and Biotechnology* **2020**, 1009.
- (2) Baig, H.; Fontanarrosa, P.; Kulkarni, V.; McLaughlin, J. A.; and Bryan Bartley, P. V.; Beal, J.; Crowther, M.; Gorochowski, T. E.; Grünberg, R.; Misirli, G.; cott Brown, J. S.; Oberortner, E.; Wipat, A.; Myers, C. J. Synthetic biology open language (SBOL) version 3.0.0. *Journal of Integrative Bioinformatics* **2020**, *17*, 20200017.
- (3) Roehner, N.; Beal, J.; Clancy, K.; Bartley, B.; Misirli, G.; Grunberg, R.; Oberortner, E.; Pocock, M.; Bissell, M.; Madsen, C., et al. Sharing structure and function in biological design with SBOL 2.0. ACS synthetic biology **2016**, 5, 498–506.
- (4) Bartley, B. A.; Choi, K.; Samineni, M.; Zundel, Z.; Nguyen, T.; Myers, C. J. .; Sauro, H. M. pySBOL: A Python Package for Genetic Design Automation and Standardization. ACS Synthetic Biology 2019, 8, 1515–1518, PMID: 30424601.
- (5) McBride, B. Handbook on ontologies; Springer, 2004; pp 51–65.
- (6) https://github.com/goksel/libSBOLj3, (accessed on June 27, 2022).
- (7) https://github.com/SynBioDex/SBOL-utilities, (accessed on June 27, 2022).
- (8) https://github.com/SynBioDex/SBOLTestSuite, (accessed on June 27, 2022).

- (9) Nguyen, T.; Walczak, N.; Sumorok, D.; Weston, M.; Beal, J. Intent Parser: A Tool for Codification and Sharing of Experimental Design. ACS Synthetic Biology 2021, 11, 502–507.
- (10) Vidal, G.; Vidal-Céspedes, C.; Rudge, T. J. LOICA: Logical Operators for Integrated Cell Algorithms. *bioRxiv* 2021,
- (11) Mante, J.; Potzsch, I.; Abam, J.; Beal, J.; Myers, C. J. Excel-SBOL converter: Creating SBOL from excel templates and vice versa. 13th International Workshop on Bio-Design Automation (IWBDA)(September 2021).
- (12) Bartley, B.; Beal, J.; Rogers, M.; Bryce, D.; Goldman, R. P.; Keller, B.; Lee, P.; Biggers, V.; Nowak, J.; Weston, M. Building an Open Representation for Biological Protocols. submitted to ACM Journal on Emerging Technologies in Computing Systems

# **TOC Graphic**

