ASSESSING PYTHON BINDINGS OF C LIBRARIES WITH RESPECT TO PYTHON IDIOMATIC CONFORMANCE

A thesis submitted to

Kent State University in partial

fulfillment of the requirements for the

Degree of Master of Computer Science

by

Joshua Behler

December 2023

Thesis written by

Joshua Behler

B.S., Kent State University, 2021

M.S., Kent State University, 2023

Approved by

_____, Advisor
Jonathan Maletic

_____, Chair, Department of Computer Science
Javed I. Khan

_____, Dean, College of Arts and Sciences
Mandy Munro-Stasiuk

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my parents Brian and Elizabeth Behler and the rest of my family for supporting me throughout my life, and always encouraging me to pursue my goals and dreams.

I would like to thank my partner, Rebecca Turner, for her continual support of me in both my professional and recreational endeavors, and for putting up with me every day.

I would like to thank my advisor, Dr. Jonathan Maletic, for providing me with such wonderful opportunities to explore academia, travel the world, and achieve things like I never have before, and also for being an extraordinary mentor. I'd also like to thank Dr. Michael Collard and Dr. Michael Decker for their support, encouragement, and wisdom.

Lastly, I'd like to thank the members of my defense committee, Dr. Gregory DeLozier, Dr. Mikhail Nesterenko, and Dr. Qiang Guan for taking time out of their busy schedules to read this thesis and hear my defense.

Joshua Behler

November 3, 2023, Kent, Ohio

CHAPTER 1

Introduction

The work presented in this thesis examines the issues facing a subset of Python modules – bindings of pre-existing C libraries. These modules differ from normal modules in that they acquire most of their implementation through the underlying C code they call and have little to no implementation added in the Python code. Because of this, they often go through widely different developmental processes than other types of Python modules. This development process is often automated, and results in cookie-cutter modules that end up providing a complex and confusing API that goes against Python standards and is often jarring to end users.

Since these bindings do not have much Python code, previously established analysis tools are not sufficient to examine these modules. Analysis programs like linters and debuggers, and more manual evaluation styles, struggle when aimed at these bindings, and end users miss out on the useful information they normally have when using other modules.

To address this, we define a set of idioms and standards focused on the implementations of Python bindings of C libraries. These standards are used to not only guide and improve development in becoming more end-user focused, but also to analyze the current state of any bindings, and how “Pythonic” they are – meaning how much they

conform to Python and community standards and are written in a way that makes use of Python features.

1.1 Motivation

While there are styling guidelines and community accepted idioms for writing Python code, there are currently no common standards defined for writing a Python module. This is especially noticeable when working with a Python binding of a C library, as the binding will often adopt coding styles from C and ignore useful Python features. Consequently, this causes any Python program that uses these modules to have a confusing, non-consistent style and an increased complexity due to having to manually handle memory in a language that does not usually allow for it.

1.2 Problem Statement

The goal of this thesis is to define a set of common Python idioms that can act as standardized guidelines for creating Python bindings of C libraries. These idioms can then be used to analyze the current state of any binding, and guide developers to improve or develop more Pythonic bindings.

1.3 Contributions

This work provides three main contributions: 11 new Pythonic idioms, evaluation of seven bindings with the idioms, and an updated, new version of the pylibsrcml module.

The 11 new idioms this thesis proposes are all defined around features needed most by Python bindings of C libraries. They are all defined intricately, with information on how to identify them and what they improve about the source code.

The 11 idioms are then used to evaluate seven different Python bindings that have been developed for popular C libraries. The idiomatic compliance and how much the binding adheres to Pythonic style is measured and compared. Additionally, the 11 idioms are also applied to the pylibsrcml binding, through a detailed development process that implements all the idioms into pylibsrcml.

1.4 Organization of the Thesis

The work is organized as follows: CHAPTER 2 includes background information on Python idioms, Python standards, and how they have been used previously. CHAPTER 3 defines the 11 new Pythonic Binding idioms, as well as the naming standards deemed relevant. CHAPTER 4 discusses the process used to analyze seven publicly available Python bindings with the idioms, and CHAPTER 5 discusses the results found from these analysis steps. CHAPTER 6 discusses how the idioms aided the updating of the pylibsrcml module, and how it compares to the previous version. CHAPTER 7 discusses future work and conclusions.

CHAPTER 2

Background and Related Work

Previous work done by Alexandru et al. gave a list of idioms that Python developers tended to agree were Pythonic in nature [Alexandru 2018]. The idioms described in their work detail what Python developers commonly consider to be good code and improve factors such as performance and readability of the code. Code that meets these standards is often called *Pythonic*, which their work also sought to define. Farooq and Zaytsev later expanded on this work by identifying 25 new idioms [Farooq and Zaytsev 2021]. They also note that due to the fast-evolving nature of Python, many idioms were previously ignored or missed because of them not being popular at the time. The paper makes specific note of the assignment expression operator (`:=`), also known as the walrus operator in Python, which at the time of the second paper had only recently been implemented and was not very common, but has since become much more widespread in modern Python code.

These idioms have inspired many different works and tools. RIdiom, a tool developed by Zhang et al., is able to automatically refactor non-idiomatic code from nine different idiomatic categories [Zhang 2023b]. Zhang et al. also analyzed the performance effects that idioms have on code, allowing developers the ability to consider how effective an idiom may be in certain situations [Zhang 2023a]. Work by Sakulniwat et al. visualized

the occurrence of the “with open” Python idiom over time, finding that usage of the idiom has increased.

In addition to idioms, other coding guidelines are important for promoting higher quality code. The Python Software Foundation maintains a set of style guides that they promote to their users as a part of their Python Enhancement Proposals (PEP). These guides aim to provide a standard to “improve the readability of code and make it consistent” [van Rossum 2013]. These articles include, but are not limited to, topics from naming conventions, stylistic formatting choices, and semantic choices for general situations. They are also often updated to add new guidelines for new features when they are released.

Beyond Python’s official style guides, there are many public guides that also advise on standardized Python formatting. Two popular ones are Google’s official Python Style Guide [Google] and The Hitchhiker’s Guide to Python [Reitz]. These guides reference the PEP guidelines often, but attempt to be wider in scope, and offer recommendations for specific situations and coding practices.

There has been little documented work on how to effectively translate a C library into a Python module. Many modules and programs exist that offer the ability to run C code in Python, including the built-in ctypes module [Python Software Foundation 2023], the Simplified Wrapper and Interface Generator program (SWIG) [SWIG 2022], and the C Foreign Function Interface module (CFFI) [Rigo and Fijalkowski]. These modules provide varying levels of automation in their bindings, with some requiring manually wrapping the C code and others generating pre-compiled Python modules that wrap the C code. However, these methods solely map C code directly to Python equivalents and lack

the ability to aggregate these features together to map to common features within Python. There exists no comprehensive list of recommended practices for adapting these bindings in a Pythonic way, and with each method providing a different interface, any one method would require wildly different steps to turn them Pythonic

CHAPTER 3

Pythonic Binding Idioms

Idioms are an important step in defining good programming practices. In 1979, Perlis and Rugaber stated that idioms are a vital step in the growth of style and structure of a programming language [Perlis and Rugaber 1979]. Python is no exception to this statement. As mentioned in Chapter 2, there is recent work done on identifying commonly used Pythonic idioms. These idioms are defined for general programming, and aid with the readability and performance of the source code. While these idioms are incredibly useful for analyzing more general case Python code, they are less helpful when looking at Python bindings. Because most, if not all, of the functionality of the module comes from the C library the binding is wrapping, most of the Python code in a binding will be focused around wrapping the C functions into Python functions and is often heavily obfuscated in pre-compiled modules. This lack of readily available implementations means that most bindings will have few or no idioms present.

For example, in Alexandru et al.'s work, one of the idioms defined is the `with` statement [Alexandru 2018]. This idiom concerns Python's context manager, which is used to handle the automatic setup and teardown of certain resources in code. Commonly, this is used for opening and closing files. An example is shown in Figure 3.1.

```
with open("file.txt",'r') as file:
    ... # Actions on file
```

```
file = open("file.txt",'r')
... # Actions on file
file.close()
```

Figure 3.1 An example of the with Python idiom (left) and functionally similar code that does not use it (right)

While a C library will often have structures that need to be opened and closed, there is no such context manager in C, meaning that an open and close function are necessary for managing data. This results in a Python binding's wrapping of said open and close functions to only call the corresponding C function, which requires the end user to manually call the open and close function. An example of an open function in Python is shown in Figure 3.2.

```
def open(self):
    crepo = ffi.new('git_repository **')
    err = C.git_submodule_open(crepo, self._subm)
    check_error(err)
    return self._repo._from_c(crepo[0], True)
```

Figure 3.2 The open function of the Submodule class in the pygit2 module. Excess indentation and comments removed for brevity

Previously defined idioms are not sufficient to analyze how Pythonic a Python binding is and help guide a developer to create a more Pythonic binding. To remedy this,

we define a set of 11 Pythonic binding idioms, which are based on previously defined idioms, but are defined specifically for Python bindings of C code. Specifically, these idioms are focused on describing what features a developer can implement and provide in their module to allow an end user to make use of the more general idioms themselves. These idioms are in general not concerned with the underlying C and Python implementation of the functionality of the module, but rather how the module author organizes and presents the functionality. Like the more generic bindings, these idioms help improve both the readability and usability of a Python binding if implemented. Table 3.1 displays a brief overview of the 11 idioms.

Idiom	Description
Iteration	Implementation of the __iter__ and/or __next__ magic functions
Context Managing	Implementation of the __enter__ and __exit__ magic functions
Casting	Implementation of any of the casting magic functions
Printability	Implementations of the __str__ and/or __repr__ magic functions
Mapping Structures to Classes	Correctly mapping structures from C code to Classes in Python code, along with mapping "attached" structure functions to class member functions
Identifying Free Functions	Correctly mapping non-"attached" C functions to free functions in Python
Destructors	Implementation of the __del__ magic function
Raising Errors	Handling integer error codes from the C code as raised Exceptions
Docstrings	The presence of docstrings on functions and classes
Annotations	The presence of annotations on functions and parameters
Selective Importing	Correctly obscuring values related to calling C within the Python module

Table 3.1 A general overview of the 11 new idioms proposed

3.1 Idioms

The following section details the in-depth definition of each idiom, how we can identify them, and some examples of how each idiom would look. The first four idioms are based on idioms defined in previous works. The latter seven are all either based on features of Python that are missed in previous literature or concerned solely with the mapping of C code to Python code. Most of these relate to the switch from the procedural programming paradigm of C to the object-oriented paradigm of Python.

3.1.1 Iteration

Iteration is Python's ability to use an object directly as the range of a loop. This idiom is based on Alexandru et al.'s previously defined idioms, "Generator expressions", "yields", and "List and Dict Comprehensions" [Alexandru 2018].

In Python, there are two main ways to make use of iteration – the for loop statement and comprehension. Both of these, which use the "for" keyword, differ from C's for loop implementation, as C uses the more common conditional and incrementation implementation, while Python only provides a for-each implementation, requiring the target of the loop to be some kind of iterable object.

```
for i in range(10):

    ...

for line in file:

    ...

args = [word for word in argv.split()]

class CustomRange:

    def __iter__(self):

        i = 0

        while i < 10:

            yield i

            i += 1

custom_arr = [num for num in CustomRange()]
```

Figure 3.3 Examples of Python iteration

To enable iteration of an object, the __iter__ function must be defined in the class. __iter__ is one of the many magic functions in Python that allows for interfacing with Python's special syntax and features. The __iter__ function is typically accompanied by the __next__ function, but this is not always true and depends on a developer's preference for implementing iteration. The yield keyword is also an option for implementing iteration, as they are indicative of a Python generator being used. Additionally, two other magic functions, __contains__ and __getitem__, can be used to implement indexing of the object. We consider these methods important and valuable for implementation of iteration

alongside __iter__ and __next__, but do not require them be present for the Iteration idiom as there is no prior evidence of them being idioms in previous work, and they are optional for iteration to work.

3.1.2 Context Managing

Context managing is Python's ability to manually handle the closing, freeing, or general teardown of a resource. Context managing is briefly mentioned at the beginning of Chapter 3, and an example of it is shown in Figure 3.1. As mentioned there, the Context Managing idiom is based on Alexandru et al.'s with statement idiom [Alexandru 2018].

Context management is used for resources that need to be explicitly opened and closed, most often on files. Using a context manager ensures that the close function will be called when the resource is done, regardless of if an error or return occurs between the open and close calls. In this regard, it is more practical and safer to use the context manager instead of manually calling the close function.

To implement context management, two functions are defined in an object's class, __enter__ and __exit__. __enter__ is called at the top of the with statement, and __exit__ is called when the scope of the with statement is left. Implementing these functions in a class will allow instances of that class to be used as the target within a with statement. An example of context management being implemented can be seen in Figure 3.4.

```
class CustomFile:
    def __enter__(self):
        ... # Call open
    def __exit__(self, type, val, tb):
        ... # Call close
```

Figure 3.4 An example implementation of context management in Python

A common implementation of __enter__ will be a sole “return self” statement, as most open or enter calls are performed during the instantiation of an object, and thus only the close needs to be implemented in __exit__.

3.1.3 Casting

In Python, the implementation of casting is done with numerous magic functions. These magic functions allow for the casting of non-default types into default types. This is useful for certain data structures that represent or hold values and offer an alternative way to get these values instead of needing to access underlying attributes or define a specific function to do so. While not explicitly defined in previous idioms lists, Alexandru et al. defined three tiers of magic functions idioms, of which the casting functions lie mostly within the “Simple” tier [Alexandru 2018].

The list of valid magic functions we consider for this idiom are: __int__, __bool__, __str__, __float__, __complex__, __index__, and __bytes__. By default, all Python objects have a __str__ function defined with a baseline behavior. Because of this, __str__

only counts for this idiom if it has a custom implementation. When one of the magic functions is implemented, they can be used by using the object as an argument in the corresponding call. An example of this with __bool__ is shown in Figure 3.5.

```
class Type:

    def __bool__(self):

        return True

a = Type()

bool_value = bool(a)
```

Figure 3.5 An example implementation and execution of boolean casting

The __index__ function is an exception to this, as there is no index() call in Python. Instead, implementing the __index__ function provides the ability to use the oct(), bin(), and hex() functions, as well as allowing it to be used alongside the slicing operator while indexing.

3.1.4 Printability

Printability is the ability of an object in Python to be used as an argument in the built-in print function. Like Casting, Printability is defined entirely though magic functions. The two magic functions important for printing are __str__ and __repr__. Having either __repr__ or __str__ is enough to satisfy the requirements of this idiom. If

both __str__ and __repr__ are implemented, print will always prefer to call the __str__ function.

As with Casting, __str__ needs to be checked for custom implementation due to all objects having a default __str__. This also means that if __str__ is implemented, both the Casting and Printability idioms are present. __repr__ does not count for Casting, as __repr__ is meant for replicating the code used to instantiate the object.

Printability is a relatively minor feature that, while not vital, is still important to help programmers trace errors they encounter and understand what their code is doing.

3.1.5 Mapping Structs to Classes

When wrapping a C library, any publicly available structures in the C code needs to be represented in the Python version as a class. Along with the structure, any C functions that are attached to the structure needs to be added as member functions of the class.

Because of C's lack of classes, we must determine a way to rule whether a function in C should be ported over as a free function or as a member function of a class. We define a function as being attached to a structure if the first parameter of the function is typed to the structure or a pointer of the structure. In general, C functions names will also indicate the structure they are acting on. This is often a good indication of which structure the function is attached to, but it is not a certainty, so checking the parameters is necessary. Some examples of C function signatures that are attached to structures are shown in Figure 3.6.

```
int getDocTitle(customDoc*);

void closeDoc(customDoc*);

int replaceLine(customDoc*,

                int, char*);
```

```
class CustomDoc:

  def get_title(self):

    ...

  def close(self):

    ...

  def replace_line(self, num, text):

    ...
```

Figure 3.6 Examples of function signatures that would be attached to the customDoc structure and their format in a Python binding

In general, a developer only ever maps C functions to a Python class if they are of the same data type, e.g., a function attached to the foo structure cannot be added to the bar class in the Python binding.

3.1.6 Identifying Free Functions

Following the standards for mapping functions that are attached to structs to classes, functions that are not attached to any structures must remain free in the binding. The parameters of these functions are most often built-in types, typedefs of built-in types, or it has no parameters. Some examples of free functions in C are shown in Figure 3.7.

Some free functions may be suited best as static class methods instead of truly free functions, such as a factory function that only constructs and returns a new instance of a structure. For the purposes of this idiom, static class methods also count as free functions.

```
char* stripSpaces(char*);

pngImage* openImage(char*);
```

Figure 3.7 Examples of free function signatures in C

3.1.7 Destructors

When programming in native Python, there is very little emphasis on manually managing the allocation and destruction of memory. All of Python's variables are references, and one may only use pass-by-reference when calling functions. While Python provides a delete statement, this statement is mostly used to remove variable identifiers from a namespace or remove elements from a collection.

To make a Python binding adhere to these practices and avoid memory leaks, any structure from the C code that has any form of a memory freeing function attached to it must be implemented as the __del__ magic function on the Python class. The __del__ function is called when an object becomes out of scope or there are no more valid identifiers that reference the object, including use of the delete statement. Attaching the freeing function to the class in this way ensures that an end user will not need to manually free an object by calling the C function and can instead let Python handle the freeing as needed. It is important to note that Python does not guarantee any order of deletion, so extra work must be put into implementing the __del__ function so that objects that depend on others are freed correctly. Similar to how the Context Manager automatically handles the closing of an object, the __del__ function must be able to close an object that is open before being

freed. This results in __del__ functions often calling multiple C functions conditionally based on the state of the object.

Because of C's more manual approach to handling memory, there are no syntactical rules for defining a memory freeing function. This means there is no concrete way to identify these functions in C solely through syntax. A common standard when creating these functions is to name them "freeFoo" or "fooFree", where foo is the name of the structure that the function will free. Manual identification of these functions is required when writing a binding and identifying this idiom.

3.1.8 Raising Errors

Python implements the ability to raise exceptions during runtime through the use of exception classes, as opposed to C's semantic implementation of integer error codes. When implementing C functions into a Python binding, any error codes need to be manually checked, and if the error code indicates an error, a corresponding exception must be raised. No function on the Python side can return an error code, which allows a Python programmer to make use of try statements to manage errors gracefully.

Because the C library will still be returning the error codes, it is still important to recognize the error codes within the binding itself, but an end user has no reason to use them. These error codes can be implemented as a list of global values, or as a custom enumeration class with entries for each error code. Similarly, exceptions can either be generalized to one main exception class that can give different error reasons, or multiple specialized exceptions for each error code.

To raise these exceptions, a simple global function can be created that checks any error code returned from a C function, and if an error is found, raise the corresponding exception. Assuming this function is also hidden from an end user, this type of function is sufficient to obstruct error codes from being returned to a user.

3.1.9 Docstrings

Python provides the ability to decorate functions and classes with a string placed as the very first statement within the block of the function or class. These strings serve a similar purpose to documentation comments – but unlike comments, they are parsed and attached to the function or class they are within. To read a docstring, access the __doc__ attribute of the class or function. If __doc__ returns None, there is no docstring attached to the object. This allows both an end user to read the information on a class or function without having to read the source code or access online documentation, as well as allowing automatic programs or linters to collect this information.

PEP 257 defines a set of style rules for docstrings [Goodger and van Rossum 2001]. These styling rules include how to format single or multi-line docstrings, how to indent them, and what information should generally be included. If any documentation comments exist in the base C code, it is permissible to copy the comment and use it as the docstring, but care must be taken to change any information that would shift, such as the type or order of parameters, return values, and function names.

While maintaining good, clear, and stylistically consistent docstrings are important, for the purposes of defining this idiom we only look for the presence of a docstring, not the content or format of the docstring itself. Figure 3.8 shows an example of a docstring.

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Figure 3.8 An example of a single-line docstring on a function, given in PEP 257 [Goodger and van Rossum 2001]

3.1.10 Annotations

Like docstrings, annotations are a way to embed information within Python source code to provide meaningful information to an end user. Also known as type hints, annotations are a way to specify typing information on functions, parameters, and variables. Annotations do not affect the code – a parameter of type string can still be passed an integer value, but they provide a way of indicating to a user what the function is expecting. It is important to note that annotations are not limited solely to names of types, and can contain any valid expression including numbers, strings, ternaries, and even lambdas. This grants flexibility to a programmer to define exactly what they would like to label their function or parameter with.

Annotations are accessible through the __annotations__ attribute on a function. This attribute returns a dictionary of parameter names to the type hint, along with a special

return entry for the function return type hint. Annotations on variables are undetectable outside of the original source code file and are ignored for this idiom.

In general, annotations can be taken directly from the C function's signature, with adjustments like changing char* to be str or bytes. However, if a Python function is implemented in a way that it could accept different types, then an appropriate union of types must be specified.

Like docstrings, a PEP style guide also exists for annotations. PEP 484 defines stylistic and contextual guidelines for how to format and what to include inside your annotations [van Rossum 2015]. For our purposes, only the presence of annotations is valued, and the style or content of the annotation is ignored.

3.1.11 Selective Importing

When making a module, it is common to group related classes, functions, and values into separate files for clearer coding. These files can be brought together through the use of import statements. However, it is important to be thoughtful when defining what is available to a user. At the top level of the module, there is an __init__.py file which is run when a user imports the module. This file is often designed to import things from other files within the module's source code to aggregate them all in one place. Anything available within the __init__.py will be available to the user initiating the import.

Python provides two main ways to control what is present within a file. The first is the ability to selectively import. Instead of writing "import module", a developer can write "from module import object". One or multiple objects can be specified, allowing for only specified things to be imported. This only works when there are a limited number of things

that need imported, and if multiple things need to be imported the import list could grow very long. The second method addresses this issue – instead of importing selective things, "from module import *" can be used to import everything within a file, and then one or multiple delete statements can be used to remove unwanted objects.

These processes work well for top level modules, but they do not apply to controlling things that are available within classes. Python does not implement a way to specify protected or private members of a class, so it is impossible to completely hide things inside of classes. To remedy this, Python's style guide specifies the use of a single leading underscore in a name to mark the values or functions as "private" [van Rossum 2013]. This standard also works for top-level module importing as well, as any name that begins with an underscore is ignored when running "from module import *"

These rules can be leveraged carefully, to ensure that an end user does not gain access to the underlying C library without a concerted effort and being aware of how the binding is structured and functions.

3.2 Naming Standards

In addition to the above idioms, an important part of writing Pythonic code is adhering to Python's standards of coding style. The PEP 8 article details an exhaustive list of how to properly style and format Python code [van Rossum 2013]. For our purposes, the naming standards are important for determining if a Python binding provides a Pythonic interface.

In this work, we refer to many different name formatting cases. The cases we use are defined here:

- camelCase: Every word after the first starts with an uppercase letter.
- PascalCase: The first letter of each compound word is capitalized. All letters of abbreviations must also be capitalized.
- snake_case: All letters within the name must be lowercase, and words separated by an underscore.
- UPPER_SNAKE_CASE: Like snake_case, but all letters must be uppercase.

When analyzing the Python binding, we evaluate the names of classes, exceptions, functions, parameters (when available), and constants. Things that we are unable to analyze without parsing the original source code, such as local variable names, are left untouched, as a normal developer would not normally interact with them.

3.2.1 Classes and Exceptions

Classes and exceptions must follow the PascalCase convention. Primitive built-in types make use of single-word lowercase names, but all other types must follow the PascalCase rule. Exceptions can be made for the use of camelCase names if a word in the name is commonly stylized to start with a lowercase letter. Class names must be kept as close as possible to the original C structure name while conforming to new style rules, such as changing the name “databaseConnection” in C to “DatabaseConnection” in Python.

3.2.2 Functions

Functions and methods must follow the snake_case convention. Function names are discouraged from mentioning the class or type they act on if they are a part of a class.

Like classes, function names must be as close as possible to the C name. An example of this is shown in Figure 3.9.

`void stackClear(stack* st);`	`def clear(self):`
	
	`    ...`

Figure 3.9 An example of changing a function name from C (left) to Python (right). The Python function is presumed to be inside a class Stack, which is why stack is removed from the name.

3.2.3 Parameters

Like functions, parameters must always follow the snake_case pattern. In addition to this, special rules for parameters in certain situations are defined in the PEP 8 Style guide:

- The first parameter of an instance method is “self.”
- The first parameter of a class method is “cls.”
- The second parameter of defined binary operator implementations is “other.”

3.2.4 Constants

Constants must follow the UPPERCASE_UNDERSCORES convention. This includes constants located inside of enumeration classes as well as global constants defined at the top of a file.

CHAPTER 4

Assessing Current Bindings

The Pythonic Binding Idioms defined above can be used to both help guide developers when creating Python bindings and evaluate the state of existing Python bindings. To highlight the contributions and effectiveness of these idioms, we detail the processes of using these idioms for both purposes. This and the following chapter detail our efforts to analyze how Pythonic current Python bindings of C libraries are through a pilot study of seven currently available bindings, alongside two additional Python modules for comparison. Chapter 5 details the evolution and development of pylibsrcml, a Python binding of the libsrcml library, using these idioms.

4.1 Collecting Libraries and Bindings

To assess how Pythonic current bindings are, a process for collecting bindings needs to be established. Our process starts with the identification of candidate C libraries to use. Using GitHub's advanced search features, we query for the most popular C code repositories that contained the phrase "lib" or "library". After identifying candidate libraries, the other repositories within the same network of the library are searched for any mention of a Pythonic wrapper or binding. In some cases, the binding is located within the main repository of the library in a sub-directory, while others are in separate repos. This

process is entirely manual and is not automated. We considered allowing unofficial bindings but decided to limit our analysis to official bindings only for simplification of our inputs, as well as leveling the field for comparison. The seven libraries gathered are detailed in Table 4.1.

C Library Name	Python Binding Name
libharu	pyharu
libvirt	libvirt
libvmi	libvmi
libvips	pyvips
libgit2	pygit2
libxml2	libxml2
libsodium	libnacl

Table 4.1 A list of the C libraries and corresponding Python bindings analyzed in this paper

Source code for the C libraries is downloaded and saved, while the Python binding modules are installed through The Python Package Index if available, and through directions on their GitHub repository if not. The Python bindings are tested to see if they can be imported by opening the Python's Integrated Development Environment and importing each library, but functionality of the library is not tested beyond this.

4.2 Analyzing Bindings for Presence of Idioms

After all the libraries and bindings are downloaded and installed, the code needs to be prepped for analysis and then actually analyzed. Each idiom requires a different set of preparation and analysis steps, each of which is detailed separately below.

4.2.1 Preparing the C Code with srcML

Before the idioms can be applied to the Python bindings, a list of function names, return types, parameters, and structure names needs to be collected for idioms that require looking at the C code for verification.

To start, the C Libraries are scanned for their include files. These are the header files that define what is made available when including a library in a C program. These header files are run through the srcML program to prepare it for analysis. srcML (www.srcML.org) is a program and an XML format that marks up the abstract syntax tree of source code while also maintaining the original code without losing textual information [Collard et al. 2013; Collard et al. 2011]. srcML allows for the easy use of normal XML tools to extract and process information, making it simple for us to use to gather information. Putting the header files through srcML allowed us to easily extract information about the structures, functions, and constants without needing to manually pour through online documentation or source code. To extract names and other information, an additional tool in the srcML Infrastructure is used: nameCollector. nameCollector is a tool that utilizes a custom SAX parser to collect every user-defined name within a srcML archive file. This tool is used to easily gather a list of names of all functions and structures within a library to easily compare to Python names. nameCollector

also provides the file location of a name, which is helpful for the analysis of parameters and return types by simplifying the process of finding the full function signature. Each library is run through srcML and then nameCollector, whose output is then saved to a comma-separated value file.

4.2.2 Preparing the Python Code

Unlike with the C code, srcML does not currently support the markup of Python. This means that analysis on the Python code must be performed through pre-existing Python analysis tools or through manual code scanning. To gather a list of relevant Python data, a small script is created which, when given the name of a module, imports the module and uses the built-in dir function to recursively traverse the publicly available modules, functions, and classes from the binding. The script then saves the following information to a JSON file:

- The names of any classes
- The names of any member functions and class attributes
- The names of any free functions
- The names of any top-level constants, as well as any class constants
- The names of any sub-modules within the top module

Each Python binding is put through this process and the resulting JSON file is saved for later processing.

4.2.3 Analyzing Name Data

After gathering all the Python names, they are categorized on whether they abide by the rules set in the PEP 8 naming standards. The quality of the names is not assessed – we solely want to analyze whether the style of the names fits within Python standards.

To check each name, a set of algorithmic rules are used for each name category as follows:

- Classes:
    - The name must begin with an uppercase letter, except in special cases of common abbreviations.
    - No underscores can be present within the name.
- Functions and Parameters:
    - The name must not contain any uppercase letters.
    - Words must be separated by underscores.
- Constants:
    - The name must not contain any lowercase letters.
    - Words must be separated by underscores.

If a given name passed all checks related to its category, then it is labeled as being within style guidelines. If it violates any rule, it is deemed as outside style guidelines, and counted against the binding being Pythonic. Because the Python source code is not analyzed when evaluating names, only parameters on functions with annotations are evaluated. Details on this are noted in Chapter 5.

4.2.4 Identifying Magic Functions

For the idioms that require the presence of one or multiple magic functions within the code, each requires an individual process for identification. This applies to five of the idioms: iteration, context managing, destructors, casting, and printability. For all the idioms, the presence of the magic function is determined by checking whether the magic function's name is present within the return of the dir function call on each class. Afterwords, additional checks must be made per idiom type.

For iteration, identification of the __iter__ or __next__ functions automatically counted the class as having the idiom. However, if the class does not have either function, we must determine if the class should have implemented the idiom by checking the original C source code. All C library functions that are attached to the corresponding structure are analyzed for the presence of a function that implies iteration, looping, or being a container. Function names that contain phrases like "next", "prev", and "iter" count for our requirements. Any Python class that has a corresponding iteration C function but does not have the __iter__ or __next__ magic functions implemented, count against how Pythonic the binding Pythonic. Those that lack both are simply ignored as being not relevant to the idiom. Figure 4.1 showcases an example of a Python class that implements the iteration idiom.

```
char* getNextLineFromFile(customDoc*);

int getNumberOfLines(customDoc*);
```

```
class CustomDoc:

    def __init__(self):

        ...

    def __iter__(self):

        for i in range(c_lib.getNumberOfLines(self.c_ptr):

            yield c_lib.getNextLineFromFile(self.c_ptr)
```

Figure 4.1 An example of a Python class in a binding that properly implements the iteration idiom.

To identify context mapping, both the __enter__ and __exit__ magic functions must be defined. Like iteration, if both the functions are present, we consider the idiom to be present. If the functions are not, we need to verify that the class/structure does not fit the idiom. All the C functions attached to the class are scanned through again, this time looking for phrases like “open”, “close”, “enter”, or “exit”. If any of these are present, the class should have the idiom, and is counted as so. Figure 4.2 shows an example of a properly implemented context manager.

```
int openDoc(customDoc*);

int closeDoc(customDoc*);
```

```
class CustomDoc:

    def __init__(self):

        ...

        c_lib.openDoc(self.c_ptr)

    def __enter__(self):

        pass # Do nothing, open called in init

    def __exit__(self):

        c_lib.closeDoc(self.c_ptr)
```

Figure 4.2 An example of a Python class in a binding that properly implements the context managing idiom

Like the previous two idioms, identifying the presence of a destructor is dependent on finding a magic function and verifying the presence of a corresponding function in the C library. For destructors, the magic function that must be identified is the __del__ function. On the C side, any function with the phrase “free” or “delete” is sufficient to count. An example of a properly implemented destructor can be seen in Figure 4.3.

```
int freeDoc(customDoc*);
```

```
class CustomDoc:

    def __init__(self):

        ...

    def __del__(self):

        c_lib.freeDoc(self.c_ptr)
```

Figure 4.3 An example of a Python class in a binding that properly implements the context managing idiom

When implementing a class that needs both context management and a destructor, the destructor must be implemented so that if the object is deleted before being closed, the close call is still made. Figure 4.4 shows an example of this.

```
class CustomDoc:

    def __init__(self):

        ...

        c_lib.openDoc(self.c_ptr)

    def __del__(self):

        if self.is_open():

            c_lib.closeDoc(self.c_ptr)

        c_lib.freeDoc(self.c_ptr)

    ... # __enter__, __exit__, and is_open
```

Figure 4.4 A merging of Figures 4.2 and 4.3, showing a class that implements a destructor that will also call the close function

Unlike the previous three idioms, analyzing attached C functions is unnecessary to prove whether an idiom applies to casting and printability. By default, all classes in Python are encouraged to be printable in some form, which also means that all classes in Python must be castable (to a string at minimum). Therefore, all that needs to be checked is whether the __str__ magic function is implemented. However, this check is more in-depth than checking the other magic functions, as all Python objects are given a default __str__ function that returns a string of the object's memory location. To verify __str__ has custom implementation, when checking the return of dir on the class, __str__ must not be of type "slot wrapper", which is what the default method is. Any deviance from this type implies a custom implementation and counts for both the casting and printability idioms.

Despite __str__ being the only magic function that would ideally require being checked for casting, because not all developers would have implemented printing, the other casting magic functions are also looked for and considered for the casting idiom. No additional checks for custom implementation need to be performed on these additional casting magic functions.

4.2.5 Analyzing Function Structure

For the remainder of the idioms, more in-depth analysis on the underlying C and Python code must be performed, making automation a difficult task. These idioms all go through a manual verification process.

As mentioned in Chapter 3, we consider a C function to be "attached" to a structure if the function's first parameter is typed to the structure. For our purposes, we consider this to be identical to the function being a member function of the structure. This classification

is necessary due to the programming paradigm difference between C and Python, C being procedural and Python being object oriented.

For these idioms, we must verify that any function implemented in the Python binding has been properly mapped over as a member function or as a free function depending on its signature in C. To accomplish this, the code of each Python function is manually examined, and any call to a C function and the Python function that made the call is recorded. For each C function call found in each Python function, the C function's signature is analyzed. For free functions, the corresponding C function's signature must not indicate attachment to a structure to remain in compliance. For member functions, the C function needs to be attached to a structure, and it needs to be attached to the same structure that the Python class is binding. Any deviation from this result in a loss of compliance.

Most functions are identifiable through this method – however, some Python modules contain pre-compiled functions, whose code is not viewable. In these cases, an educated guess is made to which C functions they call. These functions are then analyzed in the same way. For functions that have viewable source code, but make no calls to C functions, or call other functions that called C, we ignore them for idiom checking, considering them out of scope of the binding and idioms.

Checking for the presence of error handling also requires checking both the C and Python code. Using the function associations defined when checking free/member functions, the C functions' return types and documentations are manually reviewed. If the C function is determined to return an error code or some other error value, the Python

function is reviewed to see if this error code is captured and analyzed to raise an error or is ignored and passed through the call. Python functions that handled the error in some way, either by passing the error code to a specialized function that solely raises errors, or by doing in-place handling, are counted as being in-compliance with the idiom.

For docstrings and annotations, analyzing the C code is not required. Instead, only the dir function is used. For classes, the presence of the __doc__ attribute is checked for. If the value is None, or is an empty string, the idiom check fails. Otherwise, we consider the idiom as present. For functions, the same check for __doc__ is performed, alongside a check for __annotations__. If __annotations__ is present, we save the annotation information for use in checking parameter names. For both docstrings and annotations, the content of either is not taken into consideration for evaluating the presence of the idiom. We considered comparing whether the docstrings of functions matched with any documentation comments on the C function but decided against this due to documentation and requirements changing drastically when moving to Python.

Lastly, to check whether a binding has used selective importing to control what is available to an end user, is dir function is again used. We manually examined all that is brought over on a fresh import and browse for any indication of variables, functions, or sub-modules that provide direct access to C level values. If none are present, the binding passes the idiom check. Otherwise, it is counted against the module.

4.3 Analyzing Other Modules

Along with the seven bindings, two additional modules are analyzed. These modules, the built-in sqlite3 module, and the alternative XML parser module lxml, are put

through the same analysis process to catalogue what commonly considered Pythonic libraries produce and compare them to the seven main bindings we focus on. Both additional modules are also bindings of C libraries, the sqlite3 and libxml2 libraries respectively.

CHAPTER 5

Results

The results of analyzing compliance of idioms across all seven bindings are saved to individual spreadsheets for each binding, and then conglomerated here per idiom. Table 5.1 displays the summary information of total compliance for all the bindings, while Tables 5.2 through 5.9 showcase compliance per-idiom, per-binding.

Module	**Functions**		**Classes**		**Total Compliance**
	Partial	*Full*	*Partial*	*Full*	
pyharu	1 / 66	0 / 66	N/A	N/A	19.42%
libvirt	490 / 490	2 / 490	14 / 15	0 / 15	55.62%
libvmi	113 / 113	0 / 113	5 / 5	0 / 5	44.86%
pyvips	193 / 193	0 / 193	59 / 59	1 / 59	49.95%
pygit2	292 / 292	2 / 292	67 / 67	2 / 67	66.41%
libxml2	1817 / 1869	0 / 1869	31 / 43	0 / 43	49.09%
libnacl	67 / 67	0 / 67	1 / 1	0 / 1	76.24%
Total	2973 / 3090	4 / 3090	177 / 190	3 / 190	53.21%

Table 5.1 Summaries of each binding's idiomatic compliance

Table 5.1 is separated into 3 sections, Functions, Classes, and Total Compliance. For the Functions and Classes sections, two columns, Partial and Full, display ratios of compliance of each binding. The partial column shows the ratio of functions or classes within the binding that use at least one idiom to all functions/classes within the binding. The full column shows the ratio of functions/classes that use all applicable idioms to all functions/classes. The total compliance column shows the average compliance percentage for all idioms on each binding. The individual percentages for each average can be found in Tables 5.2 through 5.9.

5.1 Iteration, Destruction, and Context Mapping

Tables 5.1 through 5.3 showcase the compliance of the Iteration, Destruction, and Context Mapping idioms. Results are greatly dependent on the binding that is being analyzed. Iteration is rarely found to be needed per module, but when it is needed it is generally accounted for and included in the binding. Destructors are expectedly needed quite often but are not always implemented. This does not necessarily mean that any module that lacks compliance with the Destructor idiom will be subjected to memory issues, but this memory management is likely done in a way that adheres to C standards more than Python ones. Context management is mostly handled when needed, which is not often. A notable outlier, libxml2, has five classes that could use a context manager, but does not implement them. Like with destructors, this does not necessarily mean that libxml2 never closes their resources, but that the closing is handled in a C-like fashion.

Module	# of classes	# of Iterators implemented	# of classes that need Iterators	% compliance
pyharu	0	N/A	N/A	N/A
libvirt	15	0	0	N/A
libvmi	3	0	0	N/A
pyvips	11	0	0	N/A
pygit2	45	5	8	62.5%
libxml2	25	3	4	75.0%
libnacl	0	N/A	N/A	N/A
Total	99	8	12	75.0%

Table 5.2 Iteration idiom compliance per Python binding

Module	# of classes	# of Destructors implemented	# of classes that need Destructors	% compliance
pyharu	0	N/A	N/A	N/A
libvirt	15	13	14	92.9%
libvmi	3	0	0	N/A
pyvips	11	0	0	N/A
pygit2	45	9	28	32.1%
libxml2	25	14	20	70.0%
libnacl	0	N/A	N/A	N/A
Total	99	36	62	58.1%

Table 5.3 Destructor idiom compliance per Python binding

Module	# of classes	# of Context Managers implemented	# of classes that need Context Managers	% compliance
pyharu	0	0	0	N/A
libvirt	15	1	1	100%
libvmi	3	0	0	N/A
pyvips	11	0	1	0%
pygit2	45	1	1	100%
libxml2	25	0	5	0%
libnacl	0	0	0	N/A
Total	99	2	8	25%

Table 5.4 Context Management idiom compliance per Python binding

The pyharu and libnacl libraries are notable in that they do not implement any classes in their binding. Consequently, this means that any idiom that involves classes does not apply to them. The bindings' lack of classes is discussed in more depth in the Free and Member Functions section.

5.2 Casting and Printability

Printability and casting are mostly ignored in the bindings, with only one binding breaking 50% compliance. The number of printable classes matched identically with the number of castable classes, meaning that all classes that implemented casting likely only implemented string casting. Table 5.5 displays the results of these two idioms.

Module	# of classes	# Castable	# Printable	# Both
pyharu	0	N/A	N/A	N/A
libvirt	15	0 (0%)	0 (0%)	0 (0%)
libvmi	3	0 (0%)	0 (0%)	0 (0%)
pyvips	11	1 (9.1%)	1 (9.1%)	1 (9.1%)
pygit2	45	3 (6.7%)	3 (6.7%)	3 (6.7%)
libxml2	25	13 (52.0%)	13 (52.0%)	13 (52.0%)
libnacl	0	N/A	N/A	N/A
Total	99	17 (17.2%)	17 (17.2%)	17 (17.2%)

Table 5.5 Casting and Printability idioms compliance per Python binding

5.3 Mapping Free and Member Functions

Table 5.6 is broken into two main parts, the free functions and the member functions. For free functions, the first column lists only the number of free functions that have some form of C function call. Any free function that has no call, or is too obfuscated to tell, is ignored. The second column lists how many of those free functions have corresponding C functions that are not attached to a struct. Save for one outlier, the libraries prove very adept at mapping free functions from C to free functions in Python. pyharu and libnacl are notable, as they both only implement free functions and have no member functions. However, where they differ is in their accuracy. libnacl has 100% accuracy with its 66 free functions, while all of pyharu's 64 free functions are attached to some form of struct on the C side.

For member functions, the bindings are less consistent and changed depending on the binding being analyzed, but all maintained accuracy above 40%. Like the free

functions, the left column of the Member Functions category is the count of member functions that call any C function, with functions that do not or cannot be determined being ignored. The right column is the number of member functions that are correctly mapped to C functions that are attached to the corresponding structure. Member functions that call C functions which are attached to different structures are not counted as being correct.

Module	**Free Functions**		**Member Functions**	
	# of free functions	*# of correctly mapped free functions*	*# of member functions*	*# of correctly mapped member functions*
pyharu	64	0 (0%)	0	N/A
libvirt	19	19 (100%)	419	418 (99.8%)
libvmi	0	N/A	108	106 (98.1%)
pyvips	21	20 (95.2%)	23	10 (43.5%)
pygit2	12	12 (100%)	161	101 (62.7%)
libxml2	351	327 (93.2%)	538	238 (44.2%)
libnacl	66	66 (100%)	0	N/A
Total	533	453(85.0%)	1249	873 (69.9%)

Table 5.6 Structs to Classes and Free Function idioms compliance per binding

5.4 Raising Errors

Table 5.7 shows the compliance of functions within each binding that handle any error codes that are passed to them from the C Library. Results are mixed, with some developers being more aware of encapsulating these errors than others. Implementations of error handling also varied – some use a generic “check_error”-like function to capture any error codes and return an error if one occurred. Others do an in-place check and raise

an error inside the same function. Generally, libraries with a wide spread of error codes implement generic checking functions, while ones with few raise them in-place.

Module	# of C Functions that return Error codes	# of Python functions that raise errors
pyharu	59	0 (0%)
libvirt	155	153 (98.7%)
libvmi	78	76 (97.4%)
pyvips	0	N/A
pygit2	137	131 (99.2%)
libxml2	738	213 (28.9%)
libnacl	66	66 (100%)
Total	1233	639 (51.8%)

Table 5.7 Raising Errors idiom compliance per binding

5.5 Docstrings and Annotations

For docstrings and annotations, all classes and functions within the binding are analyzed, not just ones that are mapped from the original C code. Table 5.8 displays the compliance of all classes and functions having docstrings and annotations per binding. The bindings tended to have high percentages of function docstrings, likely due to the abundance of documentation comments within the C source code that can be mapped over.

For classes, it is roughly a coin toss on whether a class has a docstring to go with it, and of the ones that do, it is likely that most are copied documentation comments from the C code.

Annotations are virtually non-existent. Out of over 3000 functions analyzed, only 83 have annotations. The binding that has the highest compliance on annotations is libvirt, with a paltry 66 out of 490 functions annotated. While we do not analyze the contents of these annotations, it is likely that the only functions that are properly annotated are pure-Python helper functions that do not perform any C calls.

Module	# of all Classes	# of Class Docstrings	# of all Functions	# of Function Docstrings	# of Function Annotations
pyharu	0	N/A	66	0 (0%)	0 (0%)
libvirt	15	0 (0%)	490	464 (94.7%)	66 (14.2%)
libvmi	5	1 (20.0%)	113	2 (1.8%)	0 (0%)
pyvips	59	57 (96.6%)	193	185 (95.9%)	0 (0%)
pygit2	67	45 (67.2%)	292	259 (88.7%)	17 (5.8%)
libxml2	43	1 (2.3%)	1869	1646 (88.1%)	0 (0%)
libnacl	1	1 (100%)	67	65 (97.0%)	0 (0%)
Total	190	105 (55.3%)	3090	2621 (84.8%)	83 (2.6%)

Table 5.8 Docstring and Annotation idioms compliance per binding

5.6 Naming Standards

Like most of the previous results found, adherence to PEP 8's naming standards is mixed and appears to be dependent per binding. Constants showed a higher average adherence percentage than functions and classes, likely due to the similar naming standards between the two languages for constants. We do not evaluate parameters, due to the inability to access them through standard Python tools when obfuscated, the lack of available annotations for over 97% of functions, and the fact that only two out of 7 bindings implement them.

Module	Ratio of Class Names	Ratio of Function Names	Ratio of Constant Names
pyharu	N/A	1 / 66 (1.5%)	645 / 664 (97.1%)
libvirt	0 / 15 (0%)	115 / 490 (23.5%)	1168 / 1177 (99.2%)
libvmi	5 / 5 (100%)	113 / 113 (100%)	5 / 16 (31.3%)
pyvips	59 / 59 (100%)	193 / 193 (100%)	0 / 11 (0%)
pygit2	67 / 67 (100%)	291 / 292 (99.7%)	255 / 259 (98.5%)
libxml2	9 / 43 (20.9%)	216 / 1869 (11.6%)	1251 / 1251 (100%)
libnacl	1 / 1 (100%)	67 / 67 (100%)	9 / 70 (12.9%)
Total	141 / 190 (74.2%)	996 / 3090 (32.2%)	3333 / 3448 (96.7%)

Table 5.9 Naming Standard idiom compliance per binding

5.7 sqlite3, and libxml2 VS lxml

As mentioned in Chapter 4, two additional modules are analyzed using the same process as the seven previous bindings. Their results are shown in Table 5.10.

Module	**Functions**			**Classes**		
	Names	*Docs*	*Annotations*	*Names*	*Docstrings*	*Castable*
sqlite3	73 / 79	71 / 79	0 / 79	15 / 15	3 / 15	0 / 15
lxml	1154 / 1202	1135 / 1202	38 / 1202	119 / 123	107 / 123	26 / 123

Table 5.10 Summary results of the analysis of sqlite3 and lxml

Both libxml2 and lxml are modules that aim to bind the libxml2 C library. However, lxml is developed specifically to be more natively compatible with Python and "with the simplicity of a native Python API" [Welt 2022]. libxml2 however, which is officially updated alongside the C library, does not emphasize being Pythonic, noting in the official repository "some of the Python purist dislike the default set of Python bindings, rather than complaining I suggest they have a look at lxml the more pythonic bindings for libxml2 and libxslt" [Veillard and Wellnhofer 2022]. This is reflected in certain aspects of their idiomatic compliance, with lxml having a much higher percentage of names that follow the standard and docstrings. Interestingly, they both do not implement any notable amounts annotations on their functions, and libxml2 provides more casting and printability than lxml.

sqlite3 performed similarly to lxml, with almost no names not following standards, but also a surprising lack of annotations or castability. This is especially surprising, as the

sqlite3 library is a built-in module of Python. However, the module has existed in Python since Python 2, before annotations were introduced, and likely has not been updated much since its original release.

CHAPTER 6

Design and Implementation of pylibsrcml

Beyond analysis, the binding idioms, defined in this thesis, can also be used to help focus and guide development or updating of bindings to ensure they become Pythonic and simple to use for an end user. To demonstrate this, we highlight the process of updating pylibsrcml, a Python binding of the libsrcml C library.

The previous version of pylibsrcml was released in June of 2022. Since then, it has been discovered that pylibsrcml had many missing features, extraneous error checking, and a workflow that felt more like programming in C than in Python. To solve this problem and improve the binding, the following steps are taken:

- First, we identify all functions that libsrcml provides in its C API.
- Second, we group these functions up based on them being attached to a particular structure or being free.
- Third, using the rules from them mapping structs to classes and mapping free functions, we create classes based off all found structures, and add each function to its class. If a function is free, we implement it at the top level. We also follow naming guidelines for all things we port over.

- Fourth, we determine if any functions would be better suited as a one of the magic function idioms, e.g., “srcml_unit_free” is best implemented as a destructor, “srcml_archive_read_unit” works best as an iterator, etc.
- Lastly, we go through the remaining idioms and add support where possible. Each function and class are gone through and docstrings and annotations are added, taking information from the C library if necessary.

Following these steps results in pylibsrcml growing in both scale and quality. All missing features of libsrcml are implemented fully and pylibsrcml satisfies all the binding idioms to a greater degree than it did previously. Table 6.1 shows the comparison between the older version of pylibsrcml and the newer version in terms of idiomatic compliance, using the same criteria as is used for all the bindings analyzed in the previous section.

In all metrics except castability and printability, the compliance percentage has increased. The apparent decrease in castability and printability is due to the presence of four new data classes added to address missing features in pylibsrcml. Most categories also reached 100% compliance due to the development guidelines we use while developing pylibsrcml.

Because of the development process that is put in place for pylibsrcml, the process for updating pylibsrcml to match updates to libsrmcl has also been trivialized. Any time a new function or feature is added to libsrcml, the same development procedures can be used to implement these functions into pylibsrcml efficiently and in a Pythonic way.

Module	Old pylibsrcml	Current pylibsrcml
Iteration	0 / 1	2 / 2
Context Managing	0 / 1	4 / 4
Casting	1 / 2	1 / 6
Printability	1 / 2	1 / 6
Member Functions Properly Mapped	109 / 109	231 / 231
Free Functions Properly Mapped	50 / 50	50 / 50
Destructors	2 / 2	4 / 4
Raising Errors	87 [1] / 70	75 / 75
Class Docstrings	0 / 3	18 / 18
Function Docstrings	2 / 162	294 / 294
Annotations	0 / 162	270 / 294
Class Names	0 / 3	18 / 18
Function Names	154 / 162	294 / 294
Constant Names	36 / 38	39 / 39

Table 6.1 Summary results of comparing the two different version of pylibsrcml

[1] The older version of pylibsrcml included error checking on more functions that required, breaking said functions altogether.

CHAPTER 7

Conclusions and Future Work

In this thesis we present the definition and application of 11 Pythonic idioms and various guidelines intended for use on Python bindings of C libraries. As shown, these idioms prove effective for both evaluating Python bindings in terms of how Pythonic they are and guiding development of Python bindings to ensure they become Pythonic. We believe that these bindings will provide great help in standardizing Python bindings and improve how well these bindings interact with developers.

7.1.1 Future Work

In the future, we believe this work can be applied to Python modules in general to gauge how Pythonic they are. By adapting some of the idioms' rules to become broader, and not dependent on underlying C code, we can measure idiomatic compliance of all Python modules, including the Python standard library.

Work from this thesis is also inspiring a methodology in how to develop other bindings of libsrcml in other languages. Specifically, another binding of libsrcml, a JavaScript/Web Assembly wrapper tentatively known as libsrmcl.js, is being developed currently. This development is taking a similar approach to the improvements made to

pylibsrcml, with JavaScript standards being considered and implemented throughout the development process.

CHAPTER 8

References

ALEXANDRU, C.V., MERCHANTE, J.J., PANICHELLA, S., PROKSCH, S., GALL, H.C., AND ROBLES, G. 2018. On the usage of pythonic idioms. *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Association for Computing Machinery, 1–11.

COLLARD, M.L., DECKER, M.J., AND MALETIC, J.I. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, IEEE, 173–184.

COLLARD, M.L., DECKER, M.J., AND MALETIC, J.I. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. *29th IEEE International Conference on Software Maintenance (ICSM)*, 516–519.

FAROOQ, A. AND ZAYTSEV, V. 2021. There is more than one way to zen your Python. *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering*, Association for Computing Machinery, 68–82.

GOODGER, D. AND VAN ROSSUM, G. 2001. PEP 257 – Docstring Conventions | peps.python.org. https://peps.python.org/pep-0257/.

GOOGLE. Google Python Style Guide. https://google.github.io/styleguide/pyguide.html.

PERLIS, A.J. AND RUGABER, S. 1979. Programming with idioms in APL. *ACM SIGAPL APL Quote Quad 9*, 4-P1, 232–235.

PYTHON SOFTWARE FOUNDATION. 2023. ctypes — A foreign function library for Python. *Python documentation*. https://docs.python.org/3/library/ctypes.html.

REITZ, K. The Hitchhiker's Guide to Python! https://docs.python-guide.org/.

RIGO, A. AND FIJALKOWSKI, M. CFFI documentation. *CFFI 1.15.1 documentation*. https://cffi.readthedocs.io/en/latest/.

VAN ROSSUM, G., LEHTOSALO, J., AND LANGA, Ł. 2015. PEP 484 – Type Hints | peps.python.org. https://peps.python.org/pep-0484/.

VAN ROSSUM, G., WARSAW, B., AND COGHLAN, N. 2013. PEP 8 – Style Guide for Python Code | peps.python.org. https://peps.python.org/pep-0008/#function-and-variable-names.

SWIG. 2022. SWIG and Python. https://www.swig.org/.

VEILLARD, D. AND WELLNHOFER, N. 2022. Python bindings · Wiki · GNOME / libxml2 · GitLab. *GitLab*. https://gitlab.gnome.org/GNOME/libxml2/-/wikis/Python-bindings.

WELT, S. 2022. lxml - Processing XML and HTML with Python. https://lxml.de/.

ZHANG, Z., XING, Z., XIA, X., XU, X., ZHU, L., AND LU, Q. 2023a. Faster or Slower? Performance Mystery of Python Idioms Unveiled with Empirical Evidence. *Proceedings of the 45th International Conference on Software Engineering*, IEEE Press, 1495–1507.

ZHANG, Z., XING, Z., XU, X., AND ZHU, L. 2023b. RIdiom: Automatically Refactoring Non-Idiomatic Python Code with Pythonic Idioms. *Proceedings of the 45th International Conference on Software Engineering: Companion Proceedings*, IEEE Press, 102–106.