Pythonic Interfaces

Michael A. Hawker

Mikeware™

Abstract
With the evolution of computer systems, software development has become increasingly more complex. One way to deal with this increased complexity is through the use of software libraries. Many object-oriented languages, such as C++, Java, and C#, provide special constructs such as abstract classes and interfaces which ensure components are properly extended and executed. Unfortunately, the Python programming language is devoid of such features.

In this article, we present a library extension for Python to include these features into the language and allow for explicit class interfaces and abstract classes. While it has been attempted before, our library provides a simple, elegant, and Pythonic solution to the problem via a pure-Python stand-alone library. By extending the Python language in this manner, we allow developers to define concrete models for libraries and create modular code, while ensuring software system designs are enforced at run time. We also argue this provides the Python language another degree of flexibility in a formalized mechanic, as opposed to error-prone traditional “hand-shake” contracts. The usability of our proposed Python extension is demonstrated in a case study of an original game engine framework.

1 Introduction

There are many modern languages today which allow the definition of a class which does not contain implementation, but only an interface towards an implementation. This notion of an interface allows for software engineers to more easily utilize a preprogrammed library or abstract the utilization of their own library for others. It allows for inherit communication between a variety of objects without the need for objects to know explicitly how the other object is defined, only there exists a protocol established on how it can be interacted with.

When interfacing with a set of code it is desired to enforce specific rules on behavior. This eliminates the possibility of error when the rules are followed correctly, or a mechanism for debugging when rules are broken. It is through these interfaces libraries can easily be defined and extended in manners which allow them to be more flexible and easier to understand for the person wishing to utilize their functionality.

Currently, the Python language does not have such a construct. Any sort of interface implementation used now relies on verbal, error-prone, “hand-shake” contracts. If not properly documented or implemented frustrating errors can occur which have little to do with actual programming and just small programmer typos or misunderstandings. By having a formal interface and abstract class formalism, these frustrating coding and debugging issues are set aside with an initial performance overhead.

For nearly a decade now [6], there have been discussions for a formal method of class interface support in the Python Language. In our research, we have found five other varying implementations of interfaces for Python. While some may say this is plenty, and one of them must work, none of them have garnered enough attention to become a formal part of the Python language. We feel the main reason for this has been lack of focus and lack of clear, concise documentation.

There have been strong arguments to the benefits of interfaces, especially with multiple inheritance [7]. Python supports multiple inheritance well and with our library can support interfaces as well. While there have been concerns over typing changes interfaces can imply and the separation of interfaces from abstract classes [6], we can address these issues as well.

Many other high-level languages, such as Ada, C++, C#, Delphi, Eiffel, Java, PHP, and Smalltalk, include support for interface constructs. The notion of interface is not new either, being a language feature in C++, Eiffel, and Smalltalk back in the 1980s. This plethora of languages spans over a decade, and from strong to dynamically typed.

This paper presents a stand-alone pure-Python library which allows for the verification of interfaces and abstract classes. A description of the library, its features, and solutions follows. Next, an evaluation section commenting on performance and the practicality of the library as implemented in a game engine case-study. Finally, the other previously attempted solutions are addressed.

2 Pythonic Interface Library

In order to solve the many issues associated with interfaces, a complete stand-alone library solution is proposed here. With it is provided an easy concrete solution implementing not only interfaces but the notion of abstract classes as well. It can also enable type enforcement for arguments and return values of specified methods. This is all accomplished with the inclusion of the library on top of the Python v2.5 language. It has no other risks or side effects if not included. It implements the functionality of interfaces through the use of function decorators which provide a formal and familiar point of reference already utilized in the Python language. This is all accomplished in a single Python file as well, providing the simplest possible access and least amount of resistance for acceptance.

2.1 Library Feature Overview

- Multiple Inheritance
  Interface requirements checked through class hierarchy
- Extended Interfaces
  Implemented interfaces can be interfaces themselves
- Multiple Interfaces
  Class can inherit multiple interfaces
- Abstract Classes
  Only tagged methods are required for interface, others can implement specific functionality
- Multiple Behavior Models
  Provides three simple method constructs to specify interface behavior
- Simple Syntax
  A simple syntax which extends current basic language constructs
- Low Overhead
  There is a slight memory overhead for each interface created, and the only performance overhead for the simplest interface construct is added only to the instantiation of the implementing class
- Single File
  Everything is implemented in a single Python module. Nothing else is required besides the Python 2.5 base libraries. This makes it incredibly trivial to add and does not convolute itself with excess baggage.

As stated above, there are currently many complicated, fractured, and un-maintained solutions for this issue. None of them provide a simple methodology or appropriate documentation, or they focus on just the notion of interfaces without abstract classes.

In order to add interfaces to Python a simple unified front is needed. Our library meets these requirements and provides a deep benefit the Python community has been seeking for nearly a decade [6].

2.2 Simple Syntax

The library provides a very simple syntax. The interface mechanism works off of common Python constructs and mechanisms.

To create a simple interface requires only three additional things to the normal creation of a class.

The first is the interface library import:

```
from interface import interface, abstractmethod
```

The second is to define a class as normal, except to extend the interface "class":

```
class MyInterface(interface):
```

Finally, for each method in an interface, one adds the @abstractmethod decorator, exactly as one would with a @classmethod:

```
    @abstractmethod
    def required_method(self, required_argument):
        """ Interface method description or 'pass' """
```

To form the complete example:

```
from interface import interface, abstractmethod

class MyInterface(interface):
    """An Interface"""

    @abstractmethod
    def required_method(self, required_argument):
        """ 'required_method' description """
```

To utilize this interface, one simply inherits it like any other normal class:

```
class MyImplementation(MyInterface):
    def required_method(self, required_argument):
        #...my implementation...
        return True
```

If a class inheriting an interface is created without a required method an error will be thrown at instantiation. This allows for easy verification of a set of requirements on objects which would like to be used.

In order to have "implemented" the interface it is required for the method signature to be identical to the one in the interface in terms of method name and argument names, any keyword argument values are allowed to differ.

This verification is accomplished at class instantiation imposing a small overhead. This overhead lessens on additional occurrences due to caching (see Performance below).

2.3 Multiple Inheritance

One of Python's strong points is its support for multiple inheritance. By being able to extend a class with the properties of many other classes we can create new more powerful constructs from smaller individual pieces. This is also one of the main proponents and reasonings behind interfaces. Thusly, our interface library can make the most use out of Python's built-in support for multiple inheritance. Described below are scenarios in which interfaces are inherited and how the library deals with such situations.

2.3.1 Class Hierarchy

Interfaces are checked through the class hierarchy to ensure implementation. A class may be overridden multiple times and have other interfaces inherited along the way. When this occurs, only the implemented instantiated classes are checked for compliance with the underlying interfaces. If a class in between is missing implementation or is another interface, it does not matter as long as the final class has implemented all the required methods.

2.3.2 Extended Interfaces

Interfaces can be not only be inherited for implementations, but they can be inherited themselves to become new interfaces. This allows for basic constructs to be created and later extended within the form of an interface. By allowing for more complex designs such as these, it allows a programmer to be more expressive in his design to further represent what should actually occur.

2.3.3 Multiple Interfaces

One class can inherit from many interfaces. This allows for one object to interact with many other objects in a well-defined manner. By similar ideology multiple inheritance implies interfaces have been included in that extension. The programmer now has a way of creating objects which can implement multiple behaviors, such as a widget which provides a view to some data [7].

In all the cases above, overriding methods becomes an issue. Normal Python behavior is utilized in these cases. Python doesn't support multiple signatures for classes and this behavior extends to the overriding of methods in classes as well. Any concerns of method overloading and overriding in the library are therefore minimized.

2.4 Advanced Features

A very simple and effective form of interfaces has been presented. Already with it, many possibilities exist to create well engineered software libraries and programs. However, there are still even more extensions possible to provide even more functionality with very little added complexity. Introduced next will be the two additional constructs provided in the formation of interfaces and the notion of abstract classes.

2.4.1 Abstract Classes

Abstract classes provide partial implementation and interfaces which provide no implementation are often separated; however, an interface is merely an abstract class with all abstract methods. It is through this logic the notion of an abstract class need not be so different from an interface, and thusly neither should the syntax to create one.

As with initial concerns over abstract classes [6], as well as in other implementation attempts (see previous endeavors), the implementation mechanisms between interfaces and abstract classes were poorly defined. In order to be acceptable to meet these concerns, a definition would have to be clear, concise, and natural. Our library provides such an implementation clearly marking what is to be implemented by the programmer.

To create an abstract class is very straight forward. It is exactly the same as creating an interface above. The only difference is *not* to utilize the @abstractmethod decorator on the functions which are *not* part of the abstract set of the class:

```
class MyAbstractClass(interface):
        """An Abstract Class"""

        @abstractmethod
        def required_method(self):
                """Someone will have to implement this for us"""

        # Here's our implemented method
        def provided_method(self, required_argument):
                #...fetch a shrubbery...
                return 5
```

Now, it is a class which provides the child with the "provided_method" method, but still requires them to provide implementation for the "required_method".

And due to multiple inheritance as described in the previous section, it is also possible for an abstract class to inherit an interface and implement a partial set (or the whole set) of required instructions and push the abstraction to another level.

2.4.2 Loose Methods

This lends itself to the notion of a "loose" method. One which can provide its own implementation, but if overridden, must follow the specified interface:

```
class MyAbstractClass(interface):
        """An Abstract Class"""

        @abstractmethod
        def required_method(self):
                """Someone will have to implement this for us"""

        @loosemethod
        def provided_method(self, required_argument):
                #...fetch a shrubbery...
                return 5
```

Now, the "provided_method" does not have to be implemented by the inheriting class, but if it is, the enforcement of its method signature will occur.

2.4.3 Type Checking and Strict Methods

One main concern over the implementation of interfaces in Python was interfaces would bring about forced type checking [6]. However, this does not have to be the case. The purpose of an interface is to help facilitate the proper utilization of an object. Some objects inherently have strict type requirements, while others don't. Our interface library is flexible enough to support either requirement.

The @abstractmethod construct presented earlier, is used to enforce the number of arguments passed to functions within a class. The only type checking here is done at instantiation with keyword arguments. If your interface defines a keyword argument as an integer or specific value, any implementing class will have to respect the type (but not the value) for its overriding argument:

```
class MyKeywordInterface(interface):
        @abstractmethod
        def keyword_method(self, slogan, sugar=int, spice=8.11):
                """Your Ad Here"""

class MyKeywordImplementation(MyKeywordInterface):
        def keyword_method(self, slogan, sugar=10, spice=9.2):
                print "Slogan Power(" + str(sugar + spice) + "): " + slogan

key = MyKeywordImplementation()
key.keyword_method("Hello, World!", "Too Many Secrets", " aren't there?")
```

In the previous example, keyword arguments are overridden by different values, but are enforced to be of the same type i.e. sugar could not default to a string in the implementation. However, at run-time like any other Python method, no enforced type checking occurs.

Therefore, our final construct is the "strict" method. This construct provides concrete run-time argument type checking and enforces return value types as well.

```
class MyStrictClass(interface):
        """Have very specific requirements here"""

        @strictmethod
        def precise_method(self, obj, number=int, protocol=str):
                """What crazy things will happen here, I wonder?"""
                return bool
```

In the "precise_method" it is now required to pass three arguments to the method, the second of which must be of type integer, and the last to be a string. The implementer must also ensure their function returns a boolean value. If any of these conditions are not met, an error will occur. Of course, these additional checks add overhead on each function call compared to normal instantiation based checking; however, it is done transparently without disrupting documentation on the original method.

If one wants to make a "loose" strict method, then they can add our @loosemethod decorator on top of our @strictmethod one:

```
class MySomewhatStrictClass(interface):
        """Have very specific requirements here"""

        @loosemethod
        @strictmethod
        def precise_method(self, obj, number=10, protocol="standard"):
                """It will only be strict if someone overrides us, or
                   doesn't provide us with nice arguments..."""
                #...we'll provide some implementation here...
                return bool_var
                return bool
```

The implementer of our "MySomewhatStrictClass" interface can decide to utilize the built-in "precise_method" or to create his own version of it adhering to the guidelines defined within our interface. Notice the extra return call at the end of the function. This is provided to have the library adhere to the boolean type instead of an instance variable. Due to the nature of implementations, a return call can occur anywhere within the function. Rather than search and try and pick the most appropriate return call to choose, the last one is used for type inference. Since Python doesn't care if we place a redundant return call afterwards, we can utilize this exploit to provide a convenient means for type checking.

2.4.4 Special Methods: isinterface and isimplemented

In order to complete the library, two special methods have been provided for introspection purposes. They are “isimplemented” to check for interface compliance even on non-interface objects, and “isinterface” which checks if an instance of a class is an interface.

```
class strictlooseTestInterface(interface):
        """An Interface to test loose and strict methods"""

        @strictmethod
        def method0(self):
         """method 0"""

        @loosemethod
        def method1(self, arg1):
                """method 1"""
                print "1"

        @loosemethod
        @strictmethod
        def method2(self, arg1=2, arg2=4):
                """method 2"""
                return arg1+arg2
                return int

        @abstractmethod
        def method3(self):
                pass

class strictlooseTestImpl(strictlooseTestInterface):
        def method0(self):
                print "method0"

# We're not required to override this method here
#       def method2(self, arg1=5, arg2=0):
#               return arg1-arg2

        def method3(self):
                pass

print isinterface(strictlooseTestInterface)                                  # True
print isinterface(strictlooseTestImpl)                                       # False
print isimplemented(strictlooseTestInterface, strictlooseTestImpl)  # True
print strictlooseTestImpl.isimplemented(strictlooseTestInterface)   # True
print isimplemented(strictlooseTestInterface, strictlooseTestImpl())# True
print isimplemented(strictlooseTestInterface, "spamandeggs")      # False
```

The isimplemented function can work on the class object or instances of classes. There are also two ways to call the isimplemented function. If it is known the class is derived from an interface, the method based approach works well. Otherwise, the more general isimplemented function call works best (and in any situation).

2.5 Error Messages

When working with interfaces, there are a variety of concerns. To address these, a detailed error message is provided in each failure case to provide a programmer with the most appropriate feedback. Most error messages will occur on class instantiation when interfaces are actually checked. If a strict type checked method is requested, further errors can occur whenever the method is called.

The library provides a set of six error messages to describe varying issues. Each error message inherits from a base *InterfaceException* class which simple extends the built-in *Exception* class..

2.5.1 DirectInstantiationError

This error is thrown if an interface or abstract class is instantiated directly. Since the implementation is incomplete, a child class must inherit from the interface and implement all the required methods which are described by the interface.

2.5.2 RequiredMethodMissingError

The RequiredMethodMissingError is thrown on instantiation when the implementing class does not implement a required method defined in the interface. The error message includes information about which interface is not adhered to and which class was violating the interface. It also provides the interface method's documentation string and signature for quick diagnosis of the problem.

2.5.3 MismatchedSignatureError

Like the RequiredMethodMissingError above, the MismatchedSignatureError occurs if the implementing class did not adhere to the interface's definition. Similarly, the error message includes a complete set of information for rectifying the issue.

2.5.4 IncorrectArgumentTypeError

This error only occurs when the @strictmethod decorator is used. It is thrown when an implemented strict method is called if one of the arguments passed to the method does not adhere to the type specified in the interface specification. And like the other error messages, it provides a description of the offending method, the interface from which the error occurs, and the explanation of the passed argument type and what was expected.

2.5.5 IncorrectReturnTypeError

The final error which can occur is also from the @strictmethod decorator. It occurs when the method called returns a value which does not match the required specification. This will normally be the fault of the implementer as their required specification is their responsibility. But, in the case it does occur detailed information similar to the IncorrectArgumentTypeError is provided to ease debugging.

2.5.6 NotInterfaceError

The NotInterfaceError is used to prevent the usage of the special interface decorators (@abstractmethod, @loosemethod, and @strictmethod) when the class is not inheriting from the interface class. This occurs on decorator execution when the class is read by the interpreter. This prevents the breaking of the library be circumventing the checks in the base interface class.

2.6 Failure Cases

At the present time there are only two known failure cases where it is possible to circumvent the enforcement of the interface mechanism.

Interfaces are registered and checked through the Python "__new__" customization construct. If the inheriting class of an interface overrides this method for their own purposes without calling the base class or interface's __new__ method then no interface check is performed.

However, this should be a non-issue as the Python documentation itself [5], explicitly states it should be good form to call the parent class' __new__ method. Otherwise, the __new__ method should not be used. While, this is not strictly enforced by the language, it does provide the notion things are not likely to behave properly anyway if the convention is ignored.

Another possibility currently exists, which would be caused intentionally on the programmer's part. It would be to forcibly overwrite methods in the implementing class at runtime through assignment. This would circumvent the instantiation based interface checking, as the check will have already occurred.

It is possible to prevent this with Python customization methods; however, re-performing the check each time an attribute or method is set adds too much of a performance cost. It was decided for this reason to exclude the check, and make note of the possibility here.

There is also another current incompatibility with regards to the Python base types. If an attempt is made to check any built-in methods of a base type (such as remove on a list or customization functions __getitem__), with an interface that requires such a method, an error will occur. This is due to the introspection library's inability to provide information about the built-in slot wrapper objects. If this was not the case, then it would be possible to create base class templates similar to the proposal in PEP 3119 [1] for the rest of the Python 2 series. This will be discussed further in section 4.2.

2.7 Overview

We have now not only provided a very simple, low overhead interface structure, but a more concrete implementation with type checking which rivals implementations in other languages. In fact, it provides more functionality as Python has multiple inheritance. With our library, classes can have multiple interfaces, interfaces can be extended themselves, abstract classes are now possible, arguments and return values can be type checked with the @strictmethod construct, and override checking can be enforced with the @loosemethod construct.

3 Evaluation

There are multiple checks which can be performed on a library to test its usefulness. These checks go beyond the stated specifications and present concrete answers to practical questions.

The first of these tests is on actual performance of the library. While Python is not the speediest language in existence and is focused more on rapid development, it is still important to have an idea of the cost of using an additional feature. Secondly, an implemented case-study will be discussed, showing the benefits gained by adding interfaces.

3.1 Performance

One final question remaining is how much does the utilization of an interface cost in performance to a program. We cannot speak for the other solutions out there; however, they must do a similar amount of work in order to check the interface correctly. Obviously, if it was built into the language it would be faster; however, after some careful tests, it was discovered there is not a significant performance cost for the use of interfaces in relation to the time which they save in programmer effort.

Testing was performed with the built-in timeit library using 50 trials of 100000 passes each. As each computer is different, these results are used for a relational sense of cost as compared to an absolute.

First we tested normal class instantiation of a light dummy class. The average cost on this test was 0.9 usec/pass. We then created another simple class which extended the previous and tested the instantiation time of inheritance. We found with the inherited class the time to be 1.54 usec/pass, and for a double inheritance (such as an implemented interface, the interface description, followed by the base interface) the time was 2.11 usec/pass.

Secondly, an interface test was performed. We provided a simple base interface analogous to our dummy class from the previous tests. And we created a simple implementation of that interface. We first ran a test without any caching. This would force the interface to be checked at every instantiation of the implementing class. The time was about 60 usec/pass. Not very promising; however, we then enabled our caching system, thus skipping the interface check on every subsequent iteration. This yielded a substantial benefit, cutting our time to 2.69 usec/pass. Only a 1.15 usec/pass difference between interfaces and normal inheritance, or if looking at the equivalent number of inheritance steps a 0.58 usec/pass difference. The difference in times is attributed to the extra dictionary lookups needed to check if the interface has already been cached and to check for possible incorrect usage of the library. Each dictionary lookup adds approximately 0.3 usec/pass. From this standpoint our library is doing a minimal amount of extra work.

Of course, when looking at the strict type checking methods times increase dramatically as much more work is needed to ensure compliance and there is more overhead when calling the method itself as well.

Percentage wise these numbers are substantial. However, the main draw to Python is the ease and speed of development time, not run time. We believe this slight increase in instantiation time is a small price to pay for the added benefit our library provides.

3.2 Case-Study

To show the benefits of an interface library, a case-study will now be presented which shows the tribulations of working without interfaces juxtaposed with how interfaces solve simple issues. This case-study will be in the form of a game engine. In order to show the benefit of interfaces, presented here is a component to a game engine shown in both its non-interfaced original form, and the new interfaced form.

There are many types of libraries produced in today's age. They provide many functions from simple networking to complex graphical user interfaces. In each case a library implements a specific set of methods to accomplish a certain set of tasks. To make accomplishing these tasks easier, having a standardized method of handling data is helpful. Normally, the programmer is required to utilize library structures and objects to accomplish this task; however, with the power of interfaces it is possible for the programmer to provide their own implementation and for the library to act upon it.

3.2.1 The Game Engine

Game engines are becoming increasingly more popular as computational systems become more complex and as innovative games become scarce. The need to focus on design over programmer implementation is strong, especially when time pressure and failed mechanics force design changes at the last minute. By having a layer of abstraction creating games becomes a much simpler task. Having a library to complete the mundane tasks of game creations furthers creativity by allowing focus to stay on the game instead of nitpicky details.

One design for a game engine is to break apart the complex structure of a game into discrete elements. The following example makes use of a system of separated game “modes” such as menus and game parts. Each mode is registered with a base manager which controls the flow of input and the rendering stream. These system events are then passed to the active mode. In this regard, it is similar to the familiar methodology implemented in the new XNA platform [8].

Our case-study will utilize the “mode” class as an example to the benefits of interfaces and abstract classes. The mode class is registered with a base manager class which is responsible for delegating the game control. A mode is responsible for receiving various types of input, responding to music functionality, and updating the screen. Initially, before interfaces a basic mode had the following implementation:

3.2.2 Basic "Hand-shake" Protocol Based Mode

```
class Basic:
        def __init__(self, manager):
                self._manager = manager

        def activated(self, activator):
                pass

        def key_down(self, key):
                pass

        def key_up(self, key):
                pass

        def mouse_down(self, pos, button):
                pass

        def mouse_up(self, pos, button):
                pass

        def mouse_move(self, pos, buttons):
                pass

        def music_end(self):
                pass
```

This class was used as a basis for implementation to allow a mode to not be forced into providing calls to functions it may not need (e.g. if there is no mouse movement in the game then the mouse_down and mouse_up methods would not need to be implemented.), but which the game manager would attempt to call. The programmer was required also to implement a run(self, screen) method called by the manager to tell the mode to draw itself to the screen object. However, from the above implementation, it is hard to verify if the programmer has accomplished any of these tasks, especially if he does not know of them.

By utilizing the power of interfaces and abstract classes, not only can a more flexible system be created, but something which is inherently more readable too. If this simple class is broken down into a few parts, the user of the library will only make use of what they need.

In this manner a clear and concise structure can be formulated which will allow programmers to take and implement only what they need and allow whatever other part of the application they construct to determine if it can interact with their object or not. Following is the example of how this could be achieved with the basic construct which was laid out above.

3.2.3 Basic Interface Based Mode

```
from interface import interface, abstractmethod, loosemethod

class KeyboardHandlerInterface(interface):
	"""Interface for Modes requiring Keyboard input"""

	@abstractmethod
	def key_down(self, key):
		"""Handle Key Down Event"""

	@abstractmethod
	def key_up(self, key):
		"""Handle Key Up Event"""

class MouseHandlerInterface(interface):
	"""Interface for Modes requiring Mouse input"""

	@abstractmethod
	def mouse_down(self, pos, button):
		"""Handle Mouse Down Event"""

	@abstractmethod
	def mouse_up(self, pos, button):
		"""Handle Mouse Up Event"""

	@abstractmethod
	def mouse_move(self, pos, buttons):
		"""Handle Mouse Move Event"""

class MusicNotificationInterface(interface):
	"""Interface for Modes requiring Music end notification"""

	@abstractmethod
	def music_end(self):
		"""Handle Music End Event"""

class Basic(interface):
	"""A Basic Abstract Class For a General Game Mode"""

	@loosemethod
	def __init__(self, manager):
		self._manager = manager

	@abstractmethod
	def activated(self, activator):
		"""Handle When our Mode is activated"""

	@abstractmethod
	def run(self, screen):
		"""Handle Updating Drawing Screen"""
```

The programmer is now required by the interface to implement the required methods. The time was also taken to split the varying protocols into separate sections to facilitate a better hierarchy [7]. With the prior layout, the implementer is only required to implement the interfaces he requires for input recognition. If the mode does not need to receive mouse data, then it won't implement the MouseInterfaceHandler and no data will be sent. Below is an outline for a completed Mode class which is interested in keyboard input only:

```
class MyBasicMode(Basic, KeyboardInterfaceHandler):
        def activated(self, activator):
                self._my_activator = activator

        def run(self, screen):
                # draw the Flying Circus

        def key_down(self, key):
                # handle input

        def key_up(self, key):
                pass
```

3.2.4 Using an Interface Based Approach

Now, a structure is in place which provides a concrete model to follow. With the library's simple isinterface or isimplemented functions, power is provided to not only verify the interfaces on instantiation, but allowing for an easier migration from a non-interfaced framework to an interfaced one. Follows is how this migration can occur in our on-going game engine example.

Due to the special isimplemented function, the base library manager class can inspect the modes it receives and determine if they implement the correct Basic mode handler to accept the delegation of the varying game events.

The other advantage of utilizing the isimplemented function, is that it is checking for interface compatibility and not the actual class hierarchy. In this manner it is easier to convert existing projects to leverage the power of interfaces. If present classes already make use of the methods provided in the interface, they can be called and checked as if they had already implemented the interface, without actively specifying the relation. The benefit allows larger projects using protocol “hand-shake” methods to slowly convert to the use of interfaces without breaking compatibility. Therefore in the above example, if a mode implements the mouse_down, mouse_up, and mouse_move methods it will still receive those events from the game manager.

This provided flexibility extends Python's dynamic nature by not rigidly enforcing rules, but by allowing for a way to check existing classes for compatibility in a proper fashion.

While Python's dynamic nature can leverage the power of a custom built loop for each Mode based on its needs, provided next is the naïve manager loop as a final example of the game engine case-study:

Naïve Game Manager Main Loop

```
if not isimplemented(Basic, mode): raise TypeError(repr(mode) +
                                                    " must inherit from Basic Mode")

for event in event_queue():
        if event.type == QUIT:
                self._quit = True
                break
        elif self._active_mode.isimplemented(KeyboardHandlerInterface):
                if event.type == KEYDOWN:
                        self._active_mode.key_own(event.key)
                        continue
                elif event.type == KEYUP:
                        self._active_mode.key_up(event.key)
                        continue
        if self._active_mode.isimplemented(MouseHandlerInterface):
                if event.type == MOUSEBUTTONDOWN:
                        self._active_mode.mouse_down(event.pos, event.button)
                        continue
                elif event.type == MOUSEBUTTONUP:
                        self._active_mode.mouse_up(event.pos, event.button)
                        continue
                elif event.type == MOUSEMOTION:
                        self._active_mode.mouse_move(event.pos, event.buttons)
                        continue
        if event.type == MUSICEND and \
          self._music_mgr != None and \
          self._active_mode.isimplemented(MusicNotificationInterface):
                self._active_mode.music_end()

self._active_mode.run(self._screen)
```

4 Related Work

Since the initial discussions of the addition of interfaces to Python [6], much work has been done to create an interface implementation for Python. There have been many of these works, but not any one of them has succeeded in becoming a standard part of the Python language. Most other solutions have been very broad and lost focus on the simplicity of interfaces, where others are lacking proper documentation, or are using complex language constructs. Afterwards, the latest Python interface proposal will be discussed.

4.1 Previous Endeavors

Following each other interface attempt will be discussed, and an explanation of why it was unsuccessful and how our library improves upon it. In most cases these other libraries focus purely on interfaces or some complex abstraction of them.

4.1.1 PEP 245

One of the oldest solutions presented to the Python group was a Python Enhancement Proposal (PEP). PEP 245 [2] proposed an actual semantic change to the language. While the proposal was well formed, it involved much work and risk to implement and thus never moved forward in the approval process.

Our solution is an already built and tested stand-alone library, which by doing so removes all risk from its proposal. Anyone not wanting interfaces simply does not include the module. If they do, then they accept the additional functions the library adds like any other module. They are then free to utilize the power of interfaces without disrupting anything else. Also, as their interfaces will be hidden within their own module, anyone utilizing their code will not gain any additional side effects from their inclusion of the interface module (outside of the enforcement of the interfaces utilized).

4.1.2 Metaclasses

Two other "solutions" have appeared on the ASPN: Python Cookbook which are very similar. Both involve the use of metaclasses. These implementations actually do work, but are very specific about only defining interfaces, are utilizing a complex Python feature with a complex syntax, and don't verify the complete implementation of the interface.

Our solution not only provides a very simple syntax which extends a simple notion already in Python, but also enforces the correct implementation with feedback, and allows for partial implementations such as abstract classes.

4.1.3 The Zope Project

The Zope Project later defined its own interface implementation [3]. The age of this implementation is very hard to judge based on the available information; however, it definitely seems to have been conceived in thought for the longest period of time. But like most of the solutions here, an "unpythonic" method of execution is provided which also specifically focuses on interfaces alone without abstract classes.

Our solution is not part of a larger package, and provides a concise set of constructs to allow for the implementation of interfaces. There is nothing fancy or convoluted to impede the actual process of implementing an interface.

4.1.4 PyProtocols

There has also been yet another more recent endeavor called PyProtocols [4]. In concept it is the closest to our vision; however, like all the other implementations mentioned here, it lacks a clear focus and strong documentation. The PyProtocols project seems abandoned and with lack of examples, getting the out dated library to work seems impossible.

Our solution has been tested extensively on the Python 2.5 platform. It not only works well, but is very simple to understand. It is also very clear and applies a basic principle which allows for the enforcement of an entire interface or just a partial part for an abstract class.

4.2 Relation to PEP 3119

Recently, PEP 3119 [1] was released as a discussion about abstract base classes for the built-in types in Python 3000 (sets, lists, dictionaries). While the notion of built-in abstract base classes does relate to our work on interfaces (and the abstract classes it can create as well), our work applies to Python as it is now, and is more focused on the general software engineers requiring this functionality as opposed to the future change to the language discussed within PEP 3119.

PEP 3119 is also very closely related to our idealized notion of interfaces presented here. However, we have circumvented the need for metaclasses as proposed for usage within PEP 3119, making it even simpler for a programmer to implement. We have also provided other constructs for the enforcement of argument and return value type checking and for the allowance of methods which are only enforced when overridden (something which is truly unique). Our library is also geared more to general classes for complete abstracted library implementations, as opposed to the core classes of Python itself.

There has also been some discussion on PEP 3119 [9] on whether or not implementing abstract base classes would add too much "weight." And if basic interfaces should be provided which can be checked against the basic type instead of the type being an abstract class itself. Our library presented above is capable of emulating the interface of the base classes as proposed in PEP 3119 as well. However, an issue with the inspection library prevents the retrieval of detailed method information about the built-in classes, and therefore isimplemented cannot be used for sanity checks against a base class. Hopefully, in the future it would be possible to fix this issue, and be able to provide a solution in the Python 2 series for what is being proposed as a language change in Python 3000.

Ideally, our library would be incorporated into the rest of the Python v2 series and merged into any further work which develops on PEP 3119 for Python 3000.

5 References

1. PEP 3119, Introducing Abstract Base Classes
   http://www.python.org/dev/peps/pep-3119/

2. PEP 245, Python Interface Syntax
   http://www.python.org/dev/peps/pep-0245/

3. wiki.zope.org FrontPage
   http://wiki.zope.org/Interfaces/FrontPage

4. PyProtocols
   http://peak.telecommunity.com/PyProtocols.html

5. Overriding the __new__ method, Unifying types and classes in Python 2.2
   http://www.python.org/download/releases/2.2.3/descrintro/#__new__

6. Summary of the interface work of the Python types-sig
   http://www.zope.org/Members/jim/PythonInterfaces/Summary

7. Nuno Guimarães: Building Generic User Interfaces Tools: an Experience with Multiple Inheritance. In: ACM SIGPLAN Conference proceedings on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (1991) 89-96.

8. Your First Game: Microsoft XNA Game Studio Express in 2D
   http://msdn2.microsoft.com/en-us/library/bb203893.aspx

9. [Python-3000] PEP 3119 - Introducing Abstract Base Classes
   http://mail.python.org/pipermail/python-3000/2007-April/thread.html

6 Copyright

7 Author Biography

Michael A. Hawker was born in Montreal, Quebec Canada. He moved to Pennsylvania at a young age and now resides in Vermont and Montreal. He started using a computer at the age of four and started programming computers in the fourth grade. He holds a degree in Computer Science with a minor in Philosophy from McGill University and is currently pursuing a Masters in Computer Science at McGill.

In his spare time he programs games in Python while trying to start his own software company: Mikeware™. The interface library was a product of Michael's tinkering to develop a game engine wrapper around the PyGame library.