Evolutionary Prototyping: "Add Later" Static Types for Python

Roger E. Masse
Corporation for National Research Initiatives

November 17, 1998

Abstract

This paper is concerned with the benefits of prototyping as a method of evolving software and clarifying requirements. The Python type system, properties of type systems, and the aspects of "scripting" languages that lend themselves to rapid application development are reviewed, specifically dynamic typing. The benefits of static typing in "systems" programming languages and how end-products written in these languages can benefit from additional compile time checks, enhanced readability, and more efficient machine execution are also presented. Following an assertion that the Python language is well suited to prototyping tasks, an "add later" static type syntax for Python is introduced. This optional language feature would allow mature programs to acquire more conspicuous, well-understood type definitions and expand Python's capabilities towards suitability as a "systems" language implementation.

1. Introduction

The purpose of this paper is to contribute to the clarification of typing issues and their effect on software development during the various phases of the software engineering lifecycle to support a proposal for a new optional static type system for Python. The word *Type* means many things to many people ranging from the strict "*types have verifiable behavior independent of any implementation"* to the less formal "*type equals class*" definition. The type system of a language can be characterized as *strong* or *weak* and the type checking mechanism as *static* or *dynamic*. The role of a type system can be viewed in several ways [Madsen]:

1. As a means for representing concepts more explicitly in the application domain.
2. As a means for detecting certain kinds of programming errors, type errors
3. As a means for language developers to optimize for increased performance.

A type system that excels in all of these roles can have a positive effect on software systems once the implementation approaches a hardened state. In the early phases of software development however, especially when the language is used for prototyping, heavy type systems tend to interfere with flexibility and can impede productivity.

This paper does not make recommendations for any type inferencing for Python. Indeed there are aspects of the language where inferencing can be used effectively both to achieve higher performance and to generate warnings about unlikely success at run-time. Inferencing can extend beyond type safety. For example in the Python language it's quite plausible to issue a warning at compile time if the operand of an *import* statement is not found on the current module search path. Currently the import simply fails at run-time. For type safety, inferencing machinery only address goal (3) and partially goal (2) and is therefore beyond the scope of this paper.

2. Problem Statement - Python *types*

Before one can propose a new static type system for Python, one must first define what a *type* is in Python and whether the new type system will embrace the current definition or propose an entirely new one. *Type* is perhaps the most overloaded term in software language design and arguably the most contentious. For simplicity, this discussion does not include any of the several means for creating new types in Python via C or C++ extension.

Python's use of the term *type* has very little to do with the notion of *type* as a formal specification. Let us briefly analyze scenarios depicting uses of the existing type system in stock Python and tie some of the perceived shortcomings to the aforementioned roles:

1. In current Python, there are twenty-five or so *builtin* types that contain reasonably well-known type-behavior. The behavior of numbers, sequences, mappings, and mutability, for example, are well understood because they are well documented based on implementation. New types in Python are created with the *class* mechanism and so are also based on implementation. Many argue that this makes parallel development from the same "definition of the type" infeasible because the definition is based on implementation. Therefore Python is not well suited toward large-scale development efforts that involve multiple parties.
2. Python does not provide adequate mechanisms for a programmer to easily discover the type expectations of a callable entity or return value of that entity. Is the voluntary use of *assert* in the implementation sufficient? How do these assertions

get enforced in a sub-type, which overrides certain methods that contained assertions? Is it the responsibility of the developer to ensure that these assertions get copied into the new implementation? Should the caller take the responsibility (i.e. look before you leap) by using type checks of the *if hasattr(x, 'something')* form? Is it the caller's roll to ensure type safety? Often programmers are forced to read the code or rely on comments or documentation, which is often old or incorrect.

3. JPython is 10-100X slower than Java for things that really matter. Much of this speed differential can be attributed to the dynamic nature of the Python runtime. Because Java is statically typed, highly optimized byte-code can be generated since the types of the operands are known at compile time.

3. Type Systems

3.1 Strength and Safety

A type system can be classified as being strongly or weakly typed. Weakly typed systems mean that the type of an expression carries little or no information. Strongly typed languages, like PASCAL and ADA, have type signatures and behavior that can be represented in an abstract way independent of run-time support. Languages like Java, Objective-C, and Python all have runtime support for providing rich semantic information about type instances and so can be considered strongly typed.

The distinction of safety deals with the notion of when the "rich semantic information" is conveyed and how much you can reason about a type abstractly. Much type information can be inferred in Java before runtime so therefore Java has a high degree of *safety*. In Python, type information must be gleaned at runtime giving it a low degree of safety. In a perfect strongly and safely typed language, the type carries all the information about the denoted object and can be reasoned about without execution. None of the aforementioned strongly typed languages are considered perfect, pointing out that type strength and safety are relative not absolute concepts.

3.2 Static vs. Dynamic

A type system can be classified as static or dynamic. With static typing, the "type" of an attribute (e.g. and integer, a float, or an instance of a class that conforms to a particular type or set of types) is known at compile time. All identifiers associated with type referencing have explicit type information when the variable identifier is created. For example in the Java language, the statement: " int I; ", defines an integer attribute called "I". Assignments to "I" with a source having any type other than "int" will generally cause the compiler to complain. There are two important concepts at work here. 1) The type of the attribute is made explicit, and 2) the type checking is done at compile time. When a language implements dynamic typing, the type of an attribute is not evaluated until the effected line of code is executed at run-time. With dynamic typing, as is the case in the Python language, there is no explicit type assigned to an attribute identifier. If identifier "I" is assigned a string object, then I is a string, if it is assigned a list object, then I is a list. In Python and in other languages that have similar dynamic type systems, it is possible to interrogate the object at run time to authoritatively determine the current type associated with the identifier.

Static typing has the advantage of allowing the compilation phase to catch type mismatch errors up front at compile time. Static typing eliminates a specific classification of coding error; the type mismatch error, thereby making the resultant code more likely to execute without error at run-time. Static typing generally also allows the compiler of the language to generate more direct and efficient translation to the underlying machine code, which must explicitly be aware of and operate on primitive machine data-types. Finally, static typing improves the readability of programs by allowing the types to come to the surface and be more visible to the programmer. This is particularly important in maintenance.

The advantage of dynamic typing is that the developer need not add all the type-related syntax to the program making it less cluttered and easier to make type-related changes to. For example, in Python it is a trivial effort to create a function that takes a numeric argument that could be either an integer, a floating point number, or a complex number, and return the absolute value of that number in the same type that was passed in. C++ templates are a much more syntax intensive mechanism to accomplish roughly the same thing. The difference is that on average the Python function can be written much more quickly and will be less than half the size syntactically than a similarly flexible template implementation in C++. To the credit of C++, the resulting code will execute faster than the Python code. This is partially due to the fact that templates allow a variety of types to be used for the same operations while continuing to maintain the principles and advantages of static typing.

There is another variation worth mentioning which falls somewhere between the aforementioned distinction between static and dynamic typing. A language could implement explicit types when declaring attribute identifiers and at the same time, the type check could be deferred until run-time. I will refer to this combination as "explicit-dynamic" typing. This has the advantage of catching type mismatch errors immediately as they occur (at run-time) at the point of the infraction rather than later. Without explicit typing, instances need only conform to a particular set of attributes particular to the type-of-interest. Because of this, it is possible to execute well beyond a type mismatch problem only to have the error occur as some side effect later in the code.

To the developer who as become accustom to static type the dynamic approach often sounds unworkable (and vice-versa!). In reality, the majority of type problems are discovered quickly, albeit at run-time, and easily corrected. Because with dynamic typing the type errors are discovered at run-time, this approach has some disadvantage from a software reliability standpoint. Because explicit-dynamic typing does not catch errors until they occur, the software development process must rely more heavily on testing coverage tools to insure that all type mismatch related problems have been found. Explicit-dynamic typing buys little more quality assurance than does straight dynamic typing.

4. The Role of Prototyping in Software Engineering

Prototyping is good. If you accept that premise (most Python developers do), you may skip to the next section. Otherwise, this section lays the groundwork for justifying why prototyping is important.

A software engineering effort is typically divided into three main phases: requirements, design, and code. The developer can write when requirements are under-specified or ambiguous, a prototype demonstrating part or the entire requirement in question. This creates a channel of communication with the end-user. The basis of the newfound communication is derived from the understanding of how the prototype functions.

4.1 Error Distribution in the Software Lifecycle

[Neufelder] reports that in a survey of government software development projects, sixty percent of all errors were made in the requirements phase but only fifteen percent of the errors were caught in that phase. Looking at a larger cross section of software applications, she found similar results with seventy percent of errors due to requirements and design and thirty due to coding. Since software reliability is directly related to error content, a software-engineering plan that focuses on techniques to reduce requirements errors will increase the likelihood of more reliable software.

4.2 The Evolution of Lifecycle Models towards Prototyping

The Waterfall model, characterized by its rigid, cascading, one-way steps, is the oldest most familiar model. It has been blamed for causing software to be more expensive, delivered later, and to be more unreliable [Davis]. The downfall of the Waterfall model can be attributed to its document centricity and the fact that not one line of code is developed until very late in it’s high overhead process. There are no provisions in the model that allow requirements to be represented or clarified by prototyping.

Barry Boehm’s spiral model [Boehm], introduced software prototyping as a way of reducing risk. Later models such as "Evolutionary" and "Incremental Build" are based on an evolutionary prototyping rationale where the prototype is grown and refined into the final product. Some or all of the design is hardened using a prototyping language and either the final implementation is rewritten using a systems level language, or the prototype simply *becomes* the final implementation through evolution. The latter is preferable to prevent a large rewriting effort but may be necessary to increase type strength (for reliability) or for performance reasons.

4.3 Prototyping as a Tool for Requirements Analysis

Prototyping allows the designer to express an aspect of the design in the form of working code to the user. It’s a way of disambiguating functionality that may be difficult to express in words. Schneider proposes a prototyping architecture where the prototype developer is the "explainer" and the explanation receiver is the "listener". The strategy is to gradually turn this relationship into a two-way communication between equal partners. By clarifying requirements through enhanced understanding, prototyping has been adopted as a technique in software engineering to improve the accuracy of the requirements [Schneider].

5. Rapid Prototyping vs. Scripting

A rapid prototyping language should be a language that best facilitates rapid application development. This can mean many things to many people. In this section I enumerate a number of language aspects that lend themselves to rapid development.

Rapid prototyping languages share many of the same attributes as "Scripting" languages. John [Ousterhout] makes a number of assertions that he associates with scripting languages. Scripting can be thought of as a super-set of what might be done while prototyping. While prototyping focuses on rapid code development using higher level constructs, scripting focuses on rapid code development and the ability to "glue" existing components together. For example, the three most popular scripting languages: Perl, Python, and Tcl, all have well defined interfaces that allow large existing programs to be controlled and "programmed" via the native scripting language. Gluing together existing code for the purpose of demonstrating a portion of functionality can also be classified as a prototyping activity. In this way scripting languages and rapid prototyping languages are much the same.

With evolutionary prototyping, the goal is to "evolve" or "grow" some or all of the programs functionality into the final product. During this process of evolution, a software design emerges from the prototype. Often the design is specific enough to include a

final list of modules, classes, functions, and type definitions that the final design is intended to have. Either the final product will be re-implemented in a systems level language or the final product is simply the product of sufficient evolution of design in the prototyping language. With a language like Python, for example, that can be extended to provide performance and further type safety using C or C++, often the resulting software is a hybrid system containing a mix of hardened and script-like code. Care must be taken when choosing this approach to insure that the prototyping language has sufficient performance to sustain the software product in its final form and to verify that the implementation based on the prototyping language has sufficient reliability. Because of these issues, it is sometimes preferable to re-implement the final design in a systems level language despite the additional development time that must be considered [Cunningham].

6. Scripting Languages vs. Systems Programming Languages

Systems programming languages tend to be high-level (such as C, C++, ADA, or Java) and are relatively high performing. [Ousterhout] classifies systems programming languages as high-level languages that are strongly typed and are used for creating applications from scratch. I propose further clarifying this definition by adding that systems programming languages tend to be statically typed. By enhancing maintainability for large programs through readability, allowing more efficient translation to machine code, and by allowing type mismatch errors to be caught at compile time, most systems level languages embrace a static type system. The existence of large bodies of standardized class libraries that are available for C++ and Java to some degree remove the "used for creating applications from scratch" distinction as well.

Scripting languages such as Perl [Wall], Python [Lutz], Tcl [Ousterhout], and Visual Basic represent a different style of programming than systems programming languages. Scripting languages tend to assume that there already exists a collection of useful components. These components may or may not be written in different languages. Scripting languages excel in plugging together these components in a uniform maintainable way. Perl, Python, and Tcl all possess a well defined interface for accessing and otherwise controlling extension modules written in C or C++ using the native scripting language for control. In this way, scripting languages can be used to program the features of existing components.

Scripting languages tend to be concise very-high-level languages. A programming task can typically be performed with much fewer lines of code than systems programming languages. The simplicity of the code is further enhanced by the fact that most scripting languages are dynamically typed. Variable identifiers need not be type identified before use. For example in Python to create an integer value "I" with value "2" simply assign 2 to I (e.g. I = 2). To a large degree, lack of strong static type system syntax makes scripting languages more concise.

Scripting languages tend to be interpreted. This means they can demonstrate poor performance if the source code of the language is not first compiled to byte-code before execution. To muddy the waters, the Java language, which in a large body of literature is classified as a system programming language, is also interpreted. Java programs are compiled to Java byte code and then executed by the Java Virtual Machine (JVM). Because of the Java static type system, modern JVMs further compile the Java byte code into machine code. This form of byte-code to machine-code compilation is performed on-the-fly (in stages at run-time) and has been coined Just-in-time compilation. Strong and static typing combined with Just-in-time compilation have elevated Java to the realm of systems level languages.

Some scripting languages lend themselves to interactive use. Python and Tcl support interactive "shell" environments that allow programmers to test out or verify the correctness of a small segment of code. Often this is a much more direct way of insuring the appropriate syntax is being used than referring to a manual page. Interactive development can be very useful in a prototyping environment where a number of sub-program designs can be written, tested, and compared in an isolated way very quickly.

Consider the notion of Java as an example systems language at the scripting end of the spectrum and Python/JPython as an example scripting language at the systems end of the spectrum. Many assert the differences between the two are not so great. Scripting languages are becoming increasingly expressive and system languages are continuing to be implemented with more flexible run-time environments. Douglas [Cunningham] recognized the similarities and the strengths of both languages in his proposal of an evolutionary software development using both languages. In this context, Python is used as a scripting language for Java.

Although the differences between Java and Python are numerous, I argue the largest difference lies in the nature of their respective type systems. Take away this difference, and Python can participate in system level tasks on equal footing as Java while still maintaining the flexibility and expressiveness of a scripting language.

7. Python Types Revisited

While there are several approaches to creating new Python types implemented as C extension modules (stock Python C API [Van Rossum], MESS [Beaudry], ExtensionClasses [Fulton]), new data types in Python are created using inheritance. Inheritance, more specifically *implementation inheritance* is a powerful mechanism for enabling rapid development of new types and for establishing

type hierarchies based on existing implementation. The propagation of change, however, can be counter-productive, particularly if the classes are supported by different organizations [Cummins].

Strong typing restricts the flexibility of software. Coupling the type to the class ties interface to implementation. More expressive forms of typing would provide greater flexibility [Cummins], but also tend to be more complex. Type systems should provide a mechanism for validating the integration of separately developed components. Current Python can only say this is loosely true. For example in Python what does it mean to be a *file*-like object? You probably have a *read* method, but do you have *seek* or *tell*? More importantly, how do you gain confidence that an implementation of a file-like object has implemented *seek* fully without running it?

Type systems should be usable in such as way as to enhance not detract from readability of the program source. The intentions of the programmer should be made clear. The challenge for a new stronger static type system in Python is how to allow flexibility, convey readability, and provide stronger validation of types, while providing ample opportunity for language implementers to increase performance. This is a very large challenge.

8. Protocols

If one buys into the assertion that a new type system for Python must separate interface from implementation, then one might also buy into the idea that *Protocols* (as defined in Objective-C, or *interfaces* in Java) are a step in the right direction. Various Protocols discussions and proposals have been batted around the Python community for several years. These discussions are generally geared towards answering the questions of how can an object's interface be clearly defined independently of it's implementation, and what language facilities are needed to accomplish this goal? Jim Fulton (and several others prominent participants in the Python community) have long advocated a *Protocols*-based type system for Python.

Defining a *Protocol*-based type hierarchy for Python and incorporating machinery for the class to state that it implements or conforms to particular protocols would be a step towards the distinction between type and class that many feel is needed to give Python a systems language appeal. To embody the capabilities of stock Python using a protocols based type system would probably need further work in (at least) the following three areas:

- A hierarchical type system expressed in terms of protocols. Most likely this would involve subtyping and further refinement the existing builtin types (e.g. IntType, TypeType, etc) to encompass certain *behavioral* expectations expressed as protocols (e.g MUTABLE, SEEKABLE, INDEXABLE)
- Integrate the intended behavior of the builtin functions (e.g. str, coerce, max, etc)
- Embody the intended behavior of the special methods for classes (__add__, __getslice__, etc)
- The exception class hierarchy would be redefined in terms of protocols

Many believe that simply having such a protocols-based type hierarchy even without any formal way of describing behavior (other than documentation) is a strong enough premise on which to base an optional static type system. As I stated earlier, type-safety is a relative not absolute condition. Getting agreement as to how such a hierarchy should be defined is another issue. Perhaps such a type system could be modeled very closely to the way Java uses *Interfaces* to make explicit conformance to particular behavior. This would have some obvious benefits for JPython.

Others in the Python community may argue that Objective-C needs *Protocols* and Java needs *Interfaces* because they are languages without multiple inheritance.

The separation of implementation and interface is accomplished quite clearly with the *Protocols* mechanism. A clearer mapping of the existing type system in Python is probably quite possible using *Protocols*. The question remains; is the *Protocol*-based type specification enough to ensure meaningful type checking? Some may argue that *Protocols* only solve half of the type-checking problem. Do *Protocols* alone convey enough information to reasonably *ensure* that an instance really *behaves* like (for example) a file-like object even though it has the right type signature? The larger problem is how does one define and then enforce behavior in the absence of implementation.

9. Eiffel Vanilla Classes

An alternative approach to protocols that accomplishes some of the same things is exemplified by *vanilla* classes in Eiffel. These classes tend to have adjectives for names. For example, one would inherit from the standard Eiffel class COMPARABLE to announce that your class supports ordering operations (<, >, <=, >=, ==). With languages that support multiple inheritance, the first half of the type-checking problem that Protocols address can be implemented with mix-in classes and some convention. A *Protocol*-like interface could be implemented in Python in the vein of the Eiffel *vanilla* class whose name is an adjective written in all caps (for example). These special classes could be mixed-in with the implementation classes. Mix-ins have the advantage of not

forcing Python developers into such a 'split-brained' development mode that forces the separation of interface and implementation. Other may argue that the explicit separation of implementation and interface is the *whole point* of object-oriented programming through information hiding.

Vanilla classes have more potential to be used in a potential detrimental way; based on implementation. A fully implemented mix-in class does not provide a *type baseline* for independent development efforts based on the same collection of type signatures, a skeletal version of such a class, devoid of implementation, which had some optional static type semantics would. Some might argue that this is a more Pythonic way to do a *Protocols*-like mechanism. Others would counter that it muddies too much the necessary distinction between type and class and that a clearer separation of interface and implementation is needed.

The vanilla class approach does not solve the hard part of the type-checking problem. An instance could have all the proper type signatures and methods and not behave properly. Additional type specification machinery needs to be added to the vanilla class concept in order to fulfill the goals of adequate specification of type behavior independent of an implementation.

10. Design by Contract

Paul [Dubios] has long evangelized Bertrand [Meyer]'s 'design by contract' theory of program correctness. In early 1997, Paul posted an informal proposal to the readers of *comp.lang.python* for including support for some Eiffel language type semantics in Python. The proposal recommended *require* and *ensure* block support for functions as well as *invariant* clause support for classes. These constructs allow the designer to specify function pre and post conditions as well as assertions that must always remain true within the state of a class instance. The result was a compromise; the included support for *assert* as a new keyword in Python. In the absence of an explicit static type system for Python to go along with the *correctness* constructs, this was probably a good compromise.

Paul's semantic additions for assertions in Python alluded to a method for verifying that methods of the class support the intended pre and post conditions for the type. This was proposed through the use of *require* and *ensure* blocks with some additional language support.

For sub-types, the invariants of all parent types automatically apply to a newly defined type. In Eiffel, classes may not be combined if their invariants are incompatible. An assignment of the form *foo = bar* is permitted if foo and bar are instances of the same type or if bar is a descendant of foo. This supports the intuitive idea that a value of a more specialized type may be assigned to an entity of a less specialized type, but not the reverse.

The Eiffel assertion mechanisms can provide further control over the enforcement of this notion in sub-types. *Require* and *ensure* limit the amount of freedom granted to eventual redefiners. Any redefined version must satisfy a weaker or equal precondition and ensure a stronger or equal postcondition than in the original.

For code that is to be developed by different parties, a short form of the class that does not include implementation can be shared. The short form class includes only the invariant, require and ensure blocks; as well as the type signatures for function calls, and optional doc strings. It is arguable that this short form is a sort of protocol or interface

The short form class concept is Meyer's answer to the question of *why Protocols?* They represent a skeletal form of the class void of implementation. Since the short form of a class contains precondition, postcondition and invariant clauses, they also describe behavioral requirements in an unambiguous way. Meyer's concepts, exemplified in the Eiffel language, ensure correctness with these concepts combined with explicit static type declarations. Static typing is a prerequisite for design by contract.

11. "Add Later" Static Types for Python

Before proposing syntax for a static type system for Python, I believe that Python type semantics for such a system need strengthening. The current *type* semantics are more suitable for prototyping and scripting applications where Python currently excels. A new optional type system should address the perceived limitations that currently exist. I believe that a *Protocols*-based solution combined with optional require/ensure/invariant support, as found in Eiffel, would provide an improved level of type safety and program correctness as well as a clear distinction of the notions of type and class by separating interface from implementation. Perhaps an even greater precision, such as the one proposed by [Abadi] and Leino who recognized the functional logics of [Hoare] and [Floyd] do not have an analog for object-oriented programming languages, warrants further investigation.

The *Protocols* approach satisfies the advocates of separation of interface and implementation. The design-by-contract approach advocates ensuring correctness by maintaining enforceable relationships in sub-types. These two concepts are not mutually exclusive. I propose that the best solution is perhaps a combination of both approaches. For this to happen, the type-science pundits of the Python community will have to put aside their partisanship and work together toward a compromise.

Besides the benefits of improved readability and correctness, a static type system allows language designers to increase performance. Much of the 10-100X-plus performance differential that exists between (some portions of) JPython and Java could be eliminated if the Python language supported a semantic for optional static types. C-Python could also experience similar performance benefits but lacks a statically typed systems language that presents as compelling a comparison as does Java with JPython.

The following is a 'proposal-by-example' for some level of explicit static types. I make no claims to the origination of any parts. The example type system is derived to some degree from PASCAL and on previous discussions of this topic within the Python language community. I've gone so far as to propose some partial syntax for such a system because I think syntax is an important and contentious issue. I've purposely avoided some of the tricky problems associated with sub-types. It is my hope that this paper will serve as a starting (re-starting) point for further discussion and refinement of an appropriate static type system for Python and JPython.

The goals for the static type system for Python are:

- Explicit type syntax is optional. Explicit types can be added to parts or all of a particular program from the beginning, at a later time, or not at all (as is the case in standard Python).
- The changes required to the parser and other parts of the language support environment do not effect existing programs… *much*.
- Enhance not detract from the readability of the language.
- Type semantics should embody a clear distinction from class semantics.
- Type semantics should provide mechanism for defining the *behavior* of functions and classes independent of implementation
- The added type semantics/syntax would be identical in Python and JPython.
- The added static type syntax should convey enough information to allow JPython to compile to Java byte-code as efficiently (or nearly so) as Java.
- Where appropriate, maintain some level of dynamism at the expense of performance. (e.g this argument can be this type **or** this other type but nothing else)
- To the extent possible, the Python compiler will identify type mismatch errors and raise a TypeError exception at compile time. Where this is not possible, type mismatch errors will be reported as a TypeError exception at runtime.

The proposal assumes the reader is already familiar with the existing type system supported by Python. The current dynamic behavior would remain as the default. The proposal implies that importing a new built-in module "statictypes" enables all the built-in static typing machinery.

```
from statictypes import *

def myCallable : Int( i : Int, f : Float, m : myType):
        """i is an IntType, f is FloatType, m is a class instance
           derived from protocol myType. For built-in types,
           long or short versions are accepted (e.g. Int or IntType)
        """

        myInt : Int             # local variable myInt of type IntType
        myString : StringType   # local variable myString of
                                # type StringType



        myString = i            # This statement raises a
                                # TypeError at compile time

        myString = str(i)       # OK
        myInt = f               # This statement raises a
                                # TypeError at compile time
```

```
        myInt = int(f)          # OK
        return myInt
```

Figure 1

As exemplified by the preceding code fragment, optional PASCAL-like type declarations that explicitly declares the argument types, return value, and local/global variables can be partially or fully added anywhere. A new optional Protocols mechanism would be introduced via support for a new *protocol* keyword. Within a protocol definition, three new keywords would also be enabled: *require, ensure,* and *invariant*. These optional blocks would only be allowed within the protocol definition to further specify the behavior of the type where appropriate.

Further notation valid within the protocol only would most likely be appropriate, such as a reserved word *old* to reference the value of a parameter at the start of the function and a raises keyword to indicate the exception types that the implementation may raise. The assertion blocks of code would be enabled with a compiler flag and would not need to be executed once a class based on the protocol was complete and tested. The assertion mechanisms are for correctness testing only and do not contain implementation.

```
protocol Account:

        balance : FloatType
        minBalance : FloatType

        invariant:
                balance >= 0.0

        def deposit(sum : FloatType):

                require:
                        sum >= 0.0

                ensure:
                        balance = old balance + sum

        def withdraw(sum : FloatType):

                require:

                        sum >= 0.0
                        sum <= balance - minBalance

                ensure:

                        balance = old balance - sum
```

Figure 2

```
protocol Reversable:

        def reverse : SeqenceType (s : SequenceType):

                require:
                        s is not None



                ensure:
                        for i in range(len(s)):
                                old s[i] == s[len(s) - i - 1]
```

Figure 3

```
protocol ROIndexable raises (IndexError, TypeError. MyException):

        def __getitem__(key : ImutableType):

                require:
                        key is not None

protocol Indexable(ROIndexable):
        """A sub-type of ROIndexable"""

        def __setitem__( key : ImutableType, val):

                require:
                        key is not None

                ensure:
                        __getitem__(key) == val
```

Figure 4

Classes could then be *derived* from the *protocol* definitions:

```
class MyClass(Pickleable, ROIndexable):

        """User defined class MyClass conforms to the
           Pickleable and ROIndexable protocols. The
           compiler would raise a TypeError at compile
           time if all the necessary methods and signatures
           are not in place for these protocols """

        def __init__(self, i : IntType or LambdaType, m : myType):

                """i is either an IntType or a LambdaType, m is
                   either an instance based on of myType (or a
                   sub-type of myType or None). Any
                   parameter can have None as a valid type
                   unless it is disallowed via a require block.
                   Both arguments are required."""

                raise IOError                   # OK
                raise myException()             # OK
                raise myOtherException()        # Only OK if

                                                # myOtherException
                                                # is a derived type
                                                # of myException
                                                # otherwise a
                                                # compile time
                                                # TypeError is raised

                if i == 0:                      # raises TypeError at runtime
                                                # if i is LambaType
                        print "i is an intType and equals zero"
```

Figure 5

```
def myCallable : myType (self, i : Int = 0,  m : myType = None):
```

```
        """An alternative form of the argument sequence
           from previous __init__ example with default arguments."""

        if m is not None:

                # if we get here we know m is myType
                        m.mytype_callable()

        m2 : myType
        s : String = "Hello World"      # s is a StringType as
                                        # long as it lives

        i + s                           # TypeError at compile time

        s2 = s                          # s2 is a new attribute which for
                                        # the moment is a StringType

        i + s2                          # TypeError at run time

        if m2 < i :                  # TypeError at compile time
                print "m2 < i"
```

Figure 6

```
def myCallable( d : {Int : Float}, ms : [Int], ims : (Float or myType)):

        """d is a dictionary object with keys of IntType and
           values of FloatType, ms is a mutable sequence
           of IntType, ims is an immutable sequence of
           FloatType or instances (or sub-type instances)
           of myType """

        seq : ListType = [1, 2, '3', 4.0]       # An alternative way
                                                # to declare a
                                                # ListType, Nothing
                                                # is specified about
                                                # the types of the list
                                                # members

        seq2 : [[{Int : CodeType}],]            # seq2 is a list
                                                # containing lists of
                                                # dictionaries with
                                                # IntType keys and
                                                # CodeType values

        if seq == seq2:                         # Might be a TypeError at
                                                # runtime if seq does not
                                                # contain lists of dictionaries
                                                # keys th IntType keys and
                                                # CodeType values

                return
```

Figure 7

```
def myCallable( d : Mapping(Int, Float),
                ms : MutableSequence(Int),
                ims : ImmutableSequence(Float or myType)):
```

```
        """An alternative form, proposed by Jim Hugunin, of
           the previous "callable" example that is perhaps
           less readable but "Will generalize much better
           to the whole scope of parameterized interfaces
           people might want to invent". """

        # An alternative way to declare a ListType,
        # Nothing is specified about the types of the list members

        seq : ImmutableSequence = [1, 2, '3', 4.0]

        # seq2 is a list containing lists of dictionaries with
        # IntType keys and CodeType values
        seq2 : ImmutableSequence(ImmutableSequence(MappingType, CodeType)))

        if seq == seq2:
                return
```

Figure 8

```
def myFunc(d : {Int or String : Float or {Int : }} = None):

        """ d can be None (and is so by default if d is
            unspecified in the call) or a DictionaryType
            with IntType or StringType as keys and
            FloatType or DictionaryType (with IntType
            as keys and unspecified types for the values)
            as values """

        # seq contains Int or a mutable list of dictionaries with
        # Stringtype keys and values that are either myType
        # (or sub-types of myType) or TypeType or ModuleType
        seq : [ { StringType : myType or TypeType or ModuleType } or Int ]

        # Or alternatively, the above construct could be
        # represented as a series of parameterized interfaces
        seq : MutableSequence(Mapping(String, myType or Type or Module) or Int)
```

Figure 9

12. Conclusion

The concise, expressive, and dynamic nature of the Python language makes it well suited for prototyping tasks. "Add-later" static typing for Python would allow Python programs to continue to evolve into a systems level implementation once designs become hardened. Python programs could add explicit type definitions once the evolutionary phase of development completes and the interfaces of various components decided. Optionally static typed Python programs would be more readable, more maintainable, higher performing, and have more reliability because they would be based on improved type semantics and compile time checks.

I hope that this paper will serve as a starting point for the Python language designers and the Python community to reexamine the role of Python and the current Python type system as a software-engineering tool. An optional static typing system based on improved type checking semantics would broaden the appropriateness of Python and enable it to better fulfill the role of a systems language in addition to scripting and prototyping tasks that it already does well.

References

[Abadi] Martin and K.Rustan M. Leino (1998) "A logic of object-oriented programs" SRC Research Report 161, digital Equipment Corporation, Systems Reasearch Center, Palo Alto, California

[Beaudry] Don, (1994) "Deriving Built-In Classes in Python", Proceedings of the First International Python Workshop. http://www.python.org/workshops/1994-11/BuiltInClasses/BuiltInClasses_1.html (accessed 9/2/98)

[Boehm] Boehm, Barry W. (1988), "A Spiral Model for Software Development and Enhancement", IEEE Computer, 21, 61-72

[Cummings] Fred, A. (1992) "Object-Oriented Programming Languages: The Next Generation", OOPSLA 92, Addendum to the Proceedings, p81

[Cunningham] Douglas, Eswaran Subrahmanian, and Arthur Westerberg (1997), "User-Centered Evolutionary Software Development Using Python and Java" Proceedings of the 6th International Python Conference, Engineering Design Research Center, Carnegie Mellon Univeristy, Pittsburgh, PA

[Davis] Davis, A. M., Bersoff, E. H., and Comer, E. R. (1988). "A strategy for comparing alternative software development life cycle models" IEEE Transactions on Software Engineering, Vol 14, No 10, 1453-1461

[Dubois] Paul, F. (1997) "Guido's assertion proposal (Was: something silly...)" USENET news posting to comp.lang.python, March 12

[Floyd] R. W. (1967) "Assigning meanings to programs" Proceedings of the Symposium on Applied Math Vol 19, pages 19-32. American Mathematical Society, 1967.

[Fulton] Jim, (1996) "Extension Classes, Python Extension Types Become Classes" http://www.digicool.com/papers/ExtensionClass.html (accessed 9/2/98) Digital Creations, L.C

[Gosling] Gosling, James, and Henry McGilton (1995) "The Java Languages Environment", A White Paper, Sun Microsystems Computer Company, Mountain View, CA.

[Hoare] C. A. R. (1969) "An axiomatic basis for computer programming" Communications of the ACM, 12(10):576-583, October.

[Lichter] Lichter, Horst, Matthias Schneider-Hufschmidt, Heinz Zullighoven (1989), "Prototyping in Industrial Software Projects", IEEE Transactions on Software Engineering, Vol 20 No. 11, 825-832, Institute of Electrical and Electronics Engineers, Inc.

[Lutz] Lutz, Mark (1996) "Programming Python", O'Reilly and Associates, ISBN 1-56592-197-6.

[Madsen] Madsen, Ole Lehrman, Boris Magnussonm, Birger Moller-Pedersen (1990), "Strong Typing of Object-Oriented Languages Revisited" ECOOP/OOPSLA '90 Proceedings, 140-149

[Meyer] Bertrand (1998) "Building bug-free O-O software: An introduction to Design by Contract™" http://eiffel.com/doc/manuals/technology/contract/index.html (accessed 9/15/98) Interactive Software Engineering Inc.

[Neufelder] Neufelder, Ann Marie (1993) "Ensuring Software Reliability" Marcel Dekker, Inc. 270 Madison Avenue, New York, New York 10016, ISBN 0-8247-8762-5

[Schneider] Schneider, Kurt (1996) "Prototypes as Assets, not Toys – Why and How to Extract Knowledge from Prototypes", Proceedings of ISCE-18, 522-531, Institute of Electrical and Electronics Engineers, Inc.

[Van Rossum] Guido, "Python/C API Reference Manual" http://www.python.org/doc/api/api.html (accessed 9/2/98) Corporation for National Research Initiatives (CNRI)

[Wall] Wall, Larry, Tom Christiansen, R. Schwartz (1996) "Programming Perl", Second Edition, O'Reilly and Associates, ISBN: 1-56592-149-6.