Design and Implementation of an Enhanced Object-Oriented Model for the Python Interpreted Language

A Thesis

in Humanities 402

Presented To

The Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science Computer Science

by

Steven P. Miale

May 1993

On my honor I pledge that this work is my own and is done in accordance with Honor Code guidelines for papers in Humanities.

____

(Full signature)

Approved ______________________________ (Technical Advisor)
(Type Name) (Signature)

______________________________ (Humanities Advisor)
(Type Name) (Signature)

Preface

This project began approximately one year ago, as a result of a project I was working on for Dr. Pausch's research group. We had been using an interpreted language known as Tcl to create virtual reality applications. Unfortunately, one of Tcl's main drawbacks was that it was difficult to learn and use.

We began to look for a new language, and found Python. Not only was it easy to learn and use, but it was also object-oriented and easily extensible. As the system progressed, we began to modify the language, especially the object-oriented model.

This led to combining these modifications and improvements into this thesis. Many of the initial ideas had to be changed, due not only to practical coding considerations but philosophical reasons as well.

The changes I have made are currently planned to be folded into the official version of Python. The official version of Python can be transferred via FTP from ftp.cwi.nl, in the directory /pub/python.

Acknowledgements

I wish to express thanks to the following people:

Guido van Rossum, author of Python, for designing a wonderful language, helping me when I had questions, and advising me on what was practical and what was not. He not only assisted in the design process, but he also helped me to deduce some of the implementation techniques that I used.

Randy Pausch, my research and technical advisor, for allowing me to work for him for the past two years.

Robert DeLine, both as the originator of some of the design changes and user of many of the modifications made.

And, finally, to Dr. Pausch's User Interfaces Group, of which I have the honor of being a member.

Contents

Abstract

Python is an interpreted programming language that provides simple syntax, a collection of useful data types, and an object-oriented model. It was developed by Guido van Rossum at the CWI in Amsterdam.

The object-oriented model of Python was originally limited to providing methods and data objects with no protection, as well as a cumbersome syntax. This model was redesigned to correct these problems. The previous necessity of calling an instance method with the instance is no longer required. Constructors and destructors are now executed automatically. Protection for both variables and methods was added. Finally, the ability to write a class completely in C has been provided.

The new model was then implemented by changing various sections of the Python code. The modifications were then tested, some by actual inclusion into a major research project in the University of Virginia Computer Science Department. The changes outlined in this report are currently scheduled to be included with the official release of Python.

Chapter 1

Introduction

The purpose of the thesis is to improve the object-oriented design of the Python computer language by redesigning and reimplementing selected parts of the language. This was done to improve the language as a whole and to assist in University of Virginia research projects.

1.1 Project Background

Python is a language created by Guido can Rossum, a researcher at CWI, a government-sponsored research institute in the Netherlands. His description of the language is as follows:

> Python is a simple, yet powerful, interpreted programming language that bridges the gap between C and shell programming, and is thus ideally suited for "throw-away

programming" and rapid prototyping [vR93b, abstract].

One of Python's advantages over other languages of its kind is that it is object oriented. The model, like the language itself, is rather simplistic; classes and instances can be defined, and inheritance is allowed. No data protection is provided.

The simplicity of the model, however, presents many problems. First, the syntax of a class hasn't been completely refined. Each method must be defined to accept `self` as a parameter, and calls to class methods must pass self directly. Besides being cumbersome, programmers frequently forget to include `self` as a parameter to a method, thus leading to program error.

Second, there is no data protection in Python. Any data object belonging to an instance or method belonging to a class may be accessed, regardless of the nature of the requester. Third, Python does not allow either constructors or destructors, methods which are automatically called upon instantiation or destruction of an instance. Finally, Python does not allow classes to be written entirely in C. This is because a method must be a Python function rather than any type of function, Python or C.

1.2 Justification for Project

The purpose of this thesis is to fix these problems, in order to improve the Python language. In brief, the design goals are to eliminate the need for `self`, thus improving the syntax; implement data protection, constructors, and destructors; and enable classes to be written in either C or Python.

The changes are then to be incorporated into the Python language, thus providing a net benefit for Python programmers and users. Also, Python is currently used as a scripting language in the User Interfaces Group, a research team in the Department of Computer Science headed by Dr. Randy Pausch, and of which the author is a member. Many of the modifications have already been incorporated into Alice, a virtual reality toolkit which utilizes Python. [DeL93]

1.3 Overview of Report Contents

Although the design goals are straightforward, the decisions that must be made are not. Different languages may be more open or restrictive on the issue of data protection, for instance. The nature of the language itself - low or high level, interpreted or compiled - must be taken into account with respect to design.

Hence, the nature of object-oriented programming must be examined, and along with it many of the languages that have been developed and their refinement of this philosophy. This is a requirement for designing the modifications. After the design is complete, the changes will be implemented and tested.

Chapter 2

Theory of Objects

In order to redesign the Python model, a basic knowledge of the theory of object-oriented programming (also known as OOP) is required. Knowledge of other object-oriented languages is also helpful, as much can be learned from the works of others.

2.1 Basics of Object-Oriented Programming

2.1.1 Older Methodologies

Standard programming languages use what is known as process abstraction. Process abstraction is the idea that programs consist of data and functions. [Seb89, page 322] The functions then take data as parameters, and may return values or even change the data itself. What is important to note is that the functions and data are in no way connected. For instance, take the following pseudocode:

```
variable list: array[50] of integers
variable size: integer

function add_to_list(variable the_list, the_size, new_item) is
        the_size = the_size + 1
        the_list[the_size] = new_item
end

// sample code

add_to_list(list, size, 3)
```

The above example clearly presents a problem; the list and its size are contained in two different variables. This could easily lead to programmer error. Suppose the size of another list was passed to the function `add_to_list`; it would cause the program to malfunction; data could be overwritten, causing subtle glitches that would be difficult to trace. It is also somewhat cumbersome.

Then, the concept of an abstract data type was introduced. In its most simplistic form, an abstract data type is a way of organizing data into a structure which can be passed into a function. Parts of the structure can then be accessed. The above code lends itself well to this:

```
structure ListType:
        variable items: array[50] of integers
        variable size: integer
```

```
function add_to_list(variable List, new_item) is
        List.size = List.size + 1
        List.items[List.size] = new_item
end

// sample code

variable myList: ListType

add_to_list(myList, 3)
```

This is much more convenient for the programmer. However, there are still several problems with this approach. First, any portion of code has the ability to modify the internal variables of a List. This leads to decreased reliability, as a piece of code may modify the variables such that a later operation will fail. (For instance, freeing the memory used by the items array.) More importantly, however, it causes the code which accesses the internal variables to become dependent upon the implementation of a List. [Mey88, page 22] If the programmer is calling functions such as `add_to_list`, `delete_from_list`, and `sort_list`, whether a list is implemented as an array or linked list is irrelevant. If the programmer accesses the variable `List.items`, it becomes relevant.

Second, the programmer is now dependent on what type a variable is. There would be different functions to add a number to a list and to a binary tree. Finally, it is difficult to make a new type of list - perhaps one that maintains a sorted list - without writing an entirely new structure and new functions. For instance, if the add function for a sorted list was called `add_to_sorted_list`, then the programmer would have to implement a corresponding delete function, even if the delete function didn't change; otherwise, programmers would quickly become confused.

2.1.2 The Object

The fundamental concept in OOP is the notion of an object. An object is a collection of variables, which hold data, and methods, functions which operate on that data. It is similar to a structure, except that methods are also contained within the structure. Here is an example of a List object:

```
object List:
        variable items: array[50] of integers
        variable size: integer
        function add(variable new_item) is:
                List.size = List.size + 1
                List.items[List.size] = new_item
        end

// sample code

variable myList: List
myList.add(3)
```

Not only is the code more succinct, but it is easier to comprehend. List is an object which holds data - `items` and `size` - and a method called `add`. The method is now integrated with the data. The programmer is no longer requesting an operation on a piece of data - instead, it is requesting for an object to do something to itself. The main difference between the object and the previous example of an abstract data type is that objects depend on procedural abstraction, while abstract data types depend on type abstraction, or the ability to create structures. [Coo91]

2.1.3 Classes and Instances

The previous example created a single object, rather than a type of object. In object-oriented languages, there are actually two components to an object: the class, or the 'type' of an object; and the instance, a particular version of the class.

Thus, the user can define a class called `List`; it will not be an actual object, but a guideline for creating instances. Instances of the class can then be created, thus creating individual lists that can be manipulated. Each instantiation of a class has its own data, but shares methods with other instantiations of the same class.

```
class List:
        variable items: array[50] of integers
        variable size: integer
        function add(variable new_item) is:
                List.size = List.size + 1
                List.items[List.size] = new_item
        end

// sample code

variable firstList: List
variable secondList: List

firstList.add(3)
secondList.add(5)
```

`firstList` and `secondList` both have variables called `items` and `size`, as well as a method called `add`; however, the contents of the variables are different.

2.1.4 Inheritance

Inheritance is the ability to define a new class based on one or more previous classes. The new class can acquire variables and methods, and has the ability to selectively change the behavior of its "base" classes.

The list in the previous example depends entirely upon the order in which numbers were added; adding 5 and 3 would result in a list containing the number 5, followed by the number 3. An interesting idea would be to create a list which sorted the list after each addition, so that the list was always in numerical order. Instead of defining an entirely new class, SortedList could simply inherit from List.

```
class SortedList: inherit from List
        function add(variable new_item) is:
                List.add(new_item)
                sort(List.items)
        end
```

Inheritance allows programmers to reuse as much code as possible; new features can be incorporated without rewriting pre-existing code, as demonstrated above. Multiple inheritance also exists; this allows a new class to inherit from more than one class. This feature is very controversial, and is beyond the scope of this thesis.

2.1.5 Data Protection

One major problem with the List class as written is that a programmer is allowed to modify the items and size variables directly. This is not desirable; any additional functionality could easily be incorporated into either the original class or a subclass (a class which inherits from another class.) In fact, allowing the user access means that a change in the implementation of List would cause code failure.

Data protection is the ability to protect data objects and even methods from unauthorized access. The List class could be rewritten so that the variables items and size would be accessible only within the methods defined in List; nothing else would be able to access them. Not only would this increase reliability, but it also forces the programmer to view a List not as a set of variables and methods, but as an object which follows commands.

2.1.6 Abstraction

Procedural abstraction is highly desired in object-oriented programming. It means that the programmer no longer has to concern himself with the implementation of a class; rather, the programmer can simply deal with a prepackaged object that has a set of callable methods. An example is as follows:

```
class List:
        function add(item)
        function delete(item)
        function sort(item)
```

These primitives allow programmers to work at a higher level than if they were concerned about the implementation of a List, or if they were forced to call functions such as add_to_list.

2.2 Review of Object-Oriented Languages

In order to design a better model for Python, a study was made of some of the major object-oriented languages available today. They are ordered by usage; C++ and Smalltalk are the most common object-oriented languages used today, while SELF is largely a research project.

By studying these languages, it is hoped that the designer can learn from previous experiments, and thus create a higher-quality model.

2.2.1 C++

C++ is currently the most popular object-oriented language. Designed by Bjarne Stroustrup in the early to mid 1980s, it was based on C, a compiled language which enjoyed widespread popularity at the time and still does today.

In C++, data objects and methods associated with a class can have three different kinds of protection: public, wherein all have access; protected, wherein an instance of that class or any descendant class has access; and private, where only an instance of that particular class has access.

Instances have variables for not only the class it is a member of but all superclasses of that particular class. Multiple inheritance is also supported. C++ also allows methods to be overriden in subclasses, but the original method must be specified as virtual. [Str91, page 181-211]

```
class List {
        int size;
        int items[50];
        void add(int item);
}

void List::add(int item) {
        size++;
        items[size] = item;
}
```

2.2.2 Smalltalk

Smalltalk was the first major object-oriented language. It was developed in the 1970s at Xerox PARC based on ideas by Dr. Alan Kay. An interpreted language, it has undergone many revisions; Smalltalk-80 is the version discussed here. [GR83]

In Smalltalk, all program units are objects. Each object can have private (but not public) data objects as well as public methods, and is an instance of a particular class. [Seb89, pages 462-463]

```
class name              List
superclass              Object
instance variable names size
                        items
```

```
new
        ^ super new

add: item
(et cetera)
```

In some respects, Smalltalk is similar to Python. Variables are untyped; in other words, they are not predefined as integers, strings, etc., but rather are defined by way of executable code. Smalltalk, like Python, is interpreted. One main difference lies in the refinement of the OOP abilities of the two languages.

2.2.3 Objective-C

Objective-C is the second object-oriented language based on C; thus, it competed with C++. Although it was adopted as the official language by NeXT, it has not met with the success C++ has had.

Class definitions consist of two files: the interface file, which lists data objects and methods, as well as the base class if necessary; and the implementation file, which contains executable code for the methods. [PW91, pages 63-65]

```
// Interface to class List

@interface List: Object
{
        int size;
        int items[50];
}
```

```
- add: (int) item;
- delete: (int) item;
- sort;

// Implementation of class List

@implementation List
- add: (int) item
{
        size = size + 1;
        items[size] = item;
}
(et cetera)
```

Variables cannot be redefined by subclasses; methods, on the other hand, can be. Objective C is in many ways modeled on Smalltalk.

2.2.4 Eiffel

Eiffel is a compiled object-oriented language developed by Dr. Bertrand Meyer. It emphases classical software engineering techniques such as reusability, correctness, and maintenance.

Eiffel supports both public and private data objects, private being the default. All public objects must be explicitly declared using the export command. An example class is as follows:

```
class LIST export
        add, delete, sort
feature
        size: INTEGER;
        items: ARRAY[INTEGER];

        add (item: INTEGER) is
```

```
(et cetera)
```

Eiffel also allows multiple inheritance and a number of specialized features to insure program correctness, including assertions. [Mey88, pages 475-488] There is also an Eiffel variant called Sather, which sacrifices some of the software engineering techniques in order to create a more practical language. [Omo91]

2.2.5 Self

Self is an experimental language that is being developed by Sun Microsystems and Stanford University. It is unique among the languages listed here because it does not have classes or variables.

> Self is an object-oriented language for exploratory programming based on a small number of simple and concrete ideas: prototypes, slots, and behavior. Prototypes combine inheritance and instantiation to provide a framework that is simpler and more flexible than most object-oriented languages. Slots unite variables and procedures into a single construct... Finally, because Self does not distinguish state from behavior, it narrows the gaps between ordinary objects, procedures, and closures. [US87]

Because of this, and due to the rather obtuse syntax, Self has neither been widely used nor its ideas utilized in many other languages.

Chapter 3

Design

This section of the thesis will discuss the design changes for Python, as well as their rationale. The changes include the elimination of the need to pass `self`, constructor and destructor calls, data protection for both variables and methods, and the ability to construct classes entirely in C.

3.1 Incorporation of `self` with method calls

As originally designed, Python required that `self` - the instance object - be accepted as a parameter to class methods. For example:

```
class List:
        def add(self, item):
                self.items.append(item)

x = List()      # create an instance of class List
x.add(3)        # this command or
List.add(x,3)   # this one do the same thing
```

This syntax causes many problems. First, programmers frequently forget to put `self` at the beginning of a parameter tuple. This would only generate an error if `self` was accessed in the method. Otherwise, parameters would hold unexpected values; for instance, if the method add was defined as having one parameter named item, then its value when called would be a tuple with two values; the first value would be the instance (i.e. `self`) and the second value would be the number that add was called with.

Besides causing programmer error, the syntax also is rather awkward; most other OO languages automatically handle setting the value of the current instance. Finally, the reason for it being there was simply a matter of convenience when classes were first implemented. Python actually modifies the list of parameters passed to a method, putting the instance object at the beginning; when the user types the command `x.add(3)`, the add method actually gets a parameter list of `(x,3)`. Thus, `self` is required in order to know what the current instance is.

The proposed design would no longer require `self` in the list of parameters. Instead, the instance would automatically be inserted into the variable name space (the local dictionary) whenever the method is called. This would allow for the following syntax when a class is defined:

```
class List:
```

```
        def add(item):
                self.items.append(item)
```

Thus, like other object-oriented languages, the programmers would not have to concern themselves with acquiring the current instance object.

3.2 Automatic constructor calls

The previous example demonstrated how classes can be created in Python; executing `Class()` returns an instance of type `Class`. There are many cases, however, when the user wants to initialize certain variables when an instance is created. For example, when a new List is created, setting the size of the list to zero would be necessary. In Python, this can be done manually, by defining an initialization method and calling it.

```
class List:
        def init():
                self.size=0

x = List()    # create an instance of List
x.init()      # initialize it
```

One obvious problem is that the initialization method (also called a constructor) might not be called, due to programmer error. This would cause the program to fail if the constructor did something critical to the operation of the instance, such as variable initialization or opening a network connection.

A solution that most object-oriented languages, including C++ and Smalltalk, use is to allow methods with a certain name (`new` in the case of Smalltalk, or the name of the class in the case of C++) to be called automatically when an instance is created. Not only does this improve the chances for a correct program, but it is also very space efficient, requiring just one line instead of two to create an instance.

This ability would be extremely useful in Python, especially as there is no way to define variables without executing code. Thus, in the previous example, the statement `x = List()` would create an instance of class `List`, call the `init` method, and return an initialized instance. The tuple following the class name could be used as a parameter list to call `init` with.

However, since many Python classes already have a method called `init`, giving this name constructor status would break existing code. In Python, names surrounded by two underscores imply special significance; therefore, the name `__init__` was chosen. Finally, note that constructors are not necessary; if the method is not found, it is not called.

3.3 Automatic destructor calls

The case may also be made that an automatic destructor - that is, a method which is called upon deletion of an instance - is required. In many compiled languages, an instance may wish to free resources, such as memory, files, or screen space, when it is destroyed; the destructor provides a convenient way of doing this.

In Python, this is not as much of an issue. Each object in Python has a reference number; this indicates how many objects currently have access to that object. When an object's reference count becomes zero, it is destroyed. (This is how an instance is destroyed - it's count becomes zero.) However, when an instance is destroyed, all variables belonging to the instance are destroyed as well; thus, there should be no resources to free up.

However, they may still be useful, particularly for our research; when a graphical object's reference count goes to zero, it could disappear. Thus, the method `__del__`, if present, is called whenever an instance's reference count becomes zero.

3.4 Protection for data

In order to insure that programmers use the provided abstractions (for the List class, the add, delete, and sort methods), the internal variables of a class must have the ability to be protected from access by other objects. Not only will this ensure abstraction, but it will also improve reliability [Seb89, page 323] .

Languages such as C++ and Objective C allow the user to explicitly declare which data are to be private and which are to be public; this is done when the class is declared. This design proposes a similar design for Python:

```
class List:
        access size, items: private
        def add(item):
        (et cetera)
```

The `access` command is followed by any number of variables separated by commas; this list is then followed by a colon and the type of protection awarded to those variables. Types of protection include:

Public Accessible to all.

Guarded Accessible to any method in an instance.

Private Accessible only to the methods of a certain class.

In addition, read and write attributes can be given. These can be combined in any way. An example:

```
class List:
        access size, items: private, public read
        access foo: public
        def add(item):
            ...
```

In this case, the variables `size` and `items` can be read by any object, but can only be modified by the add method defined in class List. In addition, any method defined in a subclass of List will not be able to modify those two variables; this would be possible if the variables had been declared `guarded`.

3.5 Protection for methods

In addition, many OO languages allow protection for methods as well. It is occasionally desirable that a certain method not be used as an abstraction, especially if unauthorized or incorrect usage of the method in question would cause failure of the program. As the original version of Python does not support the ability to protect methods, this design proposes them.

When defining a class, a group of methods would be defined as public or private by beginning the block with `public:` or `private:`, in order to indicate the protection alloted to that block. The block would be terminated by either the termination of the class definition or the presence of another block. Public is the default protection, for reasons of backwards compatibility.

```
class List:
        def add(item):
             ...
        def delete(item):
             ...
private:
        def list-access-only():
             ...
```

In this particular case, while any object could call the methods `add` and `delete`, only the three methods of class `List` can execute the method `list-access-only`.

3.6 Class construction in C

The last planned design change is to add the ability to construct classes in C. Currently, in order to create a class which can support instances and subclassing, the class must be written entirely in Python. While this is the most common usage, it limits the extensibility of Python.

The research here, for instance, consists of many prepackaged classes; while they could be written in Python, it would be much more efficient to code them in C, since C is a compiled language. Since Python is written in C, including the portions that define classes, the modifications should be rather straightforward.

Chapter 4

Implementation and Testing

4.1 Implementation

The implementation of the entire project began in July 1992. Most of the coding was finished by December, with some maintenance taking place in the early months of 1993. Although much code was modified, most of the changes were restricted to a few relevant modules.

4.1.1 Incorporation of `self` with method calls

The implementation began on July 2nd, 1992, and was finished the next day. The only module affected was `ceval.c`, the interpreter. The modified function was `call_function`, which was executed when a function or method written in Python was called.

In previous versions of Python, if the Python function was actually an instance method, the instance was put at the beginning of the tuple passed to that method. This was the reason why `self` was required in the defined parameter list of a method.

This particular piece of code was deleted. Instead, the creation of a local dictionary was moved to the beginning of the function, and the instance object was inserted into the local dictionary under the name `self`. This simulates the old method, but does not require that self be placed into the tuple in `call_function` and removed in the method.

Maintenance took place on August 13, when it was discovered that the previous syntax for calling a class method with the current instance failed. Modifications were made to the same function, allowing it to use the current value of `self`, acquired from the local dictionary, to be passed to the class method.

4.1.2 Automatic constructor calls

The implementation of constructors was done on July 3, 1992. The modules affected included `ceval.c` and `classobject.c`, which handle all class-related functionality. The header file `classobject.h` was also changed in order to allow a new parameter to `newinstanceobject`, the function which creates instances of a particular class.

In the function `call_builtin` localed in `ceval.c`, the parameter tuple was ignored if a classobject was being called. This was changed to allow the tuple to be passed. This allowed constructor calls to have parameters. The tuple was then passed to `newinstanceobject`.

In `classobject.c`, the function `newinstanceobject` was modified to allow the additional argument. If a tuple was passed in the argument list, and the class in question defined a constructor function (a function called `__init__`), then the constructor would be called with the argument list. If there was an argument list but no constructor, an error was returned, indicating a malfunction. Also, a number of small changes were made to avoid warnings when compiling the code.

A fix was implemented on October 5, 1992, to insure that the value returned by `newinstanceobject` was not a `NULL` value. A `NULL` return value indicates a malfunction.

4.1.3 Automatic destructor calls

Destructors were implemented on November 14, 1992; the modifications were made in the file `classobject.c` to the function `instance_dealloc`. This particular function was called whenever an instance was to be deleted.

Instead of simply releasing the memory occupied by the instance, testing was done to see if a destructor (`__del__`) method was defined. If so, then it was called, then memory was freed.

One problem was that, in the destructor method, an object could create a reference to the instance; if an access to that instance occured after deallocation, a malfunction would occur. Precautions were taken to insure that an object would not be freed from memory if references to it were created; if the reference count was above zero, meaning a new reference had been created, then the object was not deleted.

4.1.4 Protection for methods

Implementation of protection for methods began in October 1992 and was completed in January 1993. Three modules were affected by the changes: `compile.c`, which transforms Python programs into a byte code; `ceval.c`, a module which executes said byte code; and `classobject.c`. The Python grammar file, `Grammar`, also had to be changed to allow for new syntax.

In file `compile.c`, the function `com_classdef` was modified to initialize a list called `privatemethods`. The intent of this list was to contain the names of all defined private methods. At the same time, a new function, `com_classmethods`, was defined. This was called when the user had defined a block of classmethods; the block could either be labeled `public` or `private`. If private, all of the function names would be added to the list `privatemethods`.

The function `build_class` was changed in the file `ceval.c`. All private functions were removed from the dictionary of methods passed to `build_class`, and were inserted in their own private dictionary. The dictionary was then passed as an additional parameter to `newclassobject`, defined in `classobject.c`. The private method dictionary was then inserted into the class data structure.

The function `class_getattr` was also modified to check the dictionaries for both public and private methods. If the method was found in the private dictionary, and the current class was the same as the class in which the private method was defined, then access was allowed;

otherwise, an access error was generated. A boundfunc object is then returned, which contains both the function and the class to which it belongs. Support for this was added into `ceval.c`.

A supplementary function to allow protection was the implementation of the `get_class` function in ĉeval.c. This works by amending the current Python frame to include the current class, as well as the current function and the local and global dictionaries. Thus, the function `eval_code` is passed an additional parameter, called `class`. This is returned when the function `get_class` is called. Also, a new type of error, `AccessError`, was introduced to indicate that an object could not be accessed because of its protection. Changes were made to `errors.h` and `bltinmdule.c`.

4.1.5 Protection for data

A preliminary design was written in August of 1992, with the final version completed in March 1993. Modules changed include `ceval.c`, `compile.c`, and `classobject.c`. Also, some of the header files were changed to allow for different function parameters, and the grammar file was changed to allow for the `access` command.

The implementation was similar to that of method protection. A dictionary called `accessinfo` was created; whenever an access command was processed in `com_access_stmt` in `compile.c`, variables with the appropriate access rights were inserted into the dictionary. The `build_class` function then passed this information to the `newclassobject` function.

Whenever a new instance was created, the instance dictionary `in_attr` was created by traversing the class heirarchy, via the function `createaccessdict`. The default value was `None`, a special value in Python.

Two additional functions had to be modified; `instance_getattr` and `instance_setattr`. Because of the changes, a lookup on the attribute dictionary no longer returned the variable, but instead an accessobject, which contains the variable, the access, and the class in which that variable was defined. Procedures called `read_access_allowed` and `write_access_allowed` were written to determine if a given variable could be accessed at that time. The first returns the object if accessible or a `NULL` if not; the second returns 1 if accessible or 0 if not.

4.1.6 Class construction in C

The implementation was finished on September 5, 1992. Two files were modified: `ceval.c` and `classobject.c`, in addition to the header file `classobject.h`.

The modification consisted of adding a new object called an `instancebuiltinobject`. Similar to an `instancemethodobject`, it handles the case in which a method is a C function, whereas the instancemethod only accepts Python methods. First, acceptance for that particular object was inserted into the `call_builtin` function in `ceval.c`. Then, changes were made to `classobject.c`; these include defining the `instancebuiltinobject` and returning one if a class method is found to be a C function.

A minor fix was implemented on October 12, to allow for the passing of `self` as the first parameter to a C function, the other parameter being the argument list. This was accomplished in `call_builtin`.

4.2 Testing

This section documents some of the testing of the changes made to the Python language. Not all tests are included in this report; extensive testing took place during implementation. Instead, tests were selected in order to show a correct basic implementation.

In this section, results directly from the Python interpreter have been inserted into the document. While the author does not wish to document the entire language in this report, it is helpful to know that a line beginning with the three characters `>>>` mean the beginning of a block, while the prompt `...` implies continuation of an above block. This is an artifact of the Python interpreter, rather than an aspect of the language itself.

Finally, note that all changes except the protection changes and the destructor have been incorporated into the Alice system, which is being used by a large subset of the User Interfaces group. Few problems have been encountered, and all of those errors were corrected early in the beta-testing process.

4.2.1 Incorporation of `self` with method calls

In order to test the functionality of this modification, several tests were used. The initial test consisted of a simple call to a class with the new method, one which was successful.

```
>>> class myclass:
...     def set(x):
...             self.x=x
...     def get():
...             return self.x
...
>>> inst=myclass()
>>> inst.set(10)
>>> inst.get()
10
```

The next test was to insure that a class method could be called within another class. To do this, a subclass was created which called its parent class.

```
>>> class baseclass(myclass):
...     def set(x):
...             print 'setting to ',x
...             myclass.set(x)
...     def get():
...             x=myclass.get()
...             print 'got ',x
...
>>> inst=baseclass()
>>> inst.set(5)
setting to  5
>>> inst.get()
got  5
```

4.2.2 Automatic constructor calls

The sole test was to create a class with a constructor, call it, and test to see if the variables set by the constructor took place; it was successful.

```
>>> class myclass:
...     def __init__(x):
...             self.x=x
...     def get():
...             return self.x
...
>>> inst=myclass(3)
>>> inst.get()
3
```

4.2.3 Automatic destructor calls

For this modification, two tests were run. The first test set a global variable when the `__del__` method was called; this allowed easy verifiability. The second test was to insure that an instance was not destroyed if the destructor made a copy of that instance. Both were successful.

```
>>> class a:
...     def __del__():
...             global x
...             x=3
...
>>> y=a()
>>> del(y)
>>> x
3
>>> class b:
...     def __del__():
...             global x
...             x=self
...
>>> y=b()
>>> del(y)
>>> x.x=3
>>> x
<instance object at 4c958>
```

4.2.4 Protection for methods

Two tests were run for this modification. The first test consisted of a class with public and private methods; the public methods called the private methods. From the prompt, an instance could call the public methods, but not the private ones.

```
>>> class myclass:
...     def public_method():
...             print 'in public method'
...             self.private_method()
... private:
...     def private_method():
...             print 'in private method'
...
>>> inst=myclass()
>>> inst.public_method()
in public method
in private method
>>> inst.private_method()
AccessError: private_method
Stack backtrace (innermost last):
  File "<stdin>", line 1
```

The second test was to insure that a descendant of a class containing a private method also could not access it; it also proved successful, generating an `AccessError` after an attempt to call a private method.

```
>>> class subclass(myclass):
...     def main():
...             print 'trying public methods'
...             self.public_method()
...             print 'trying private methods'
```

```
...             self.private_method()
...
>>> inst=subclass()
>>> inst.main()
trying public methods
in public method
in private method
trying private methods
AccessError: private_method
Stack backtrace (innermost last):
  File "<stdin>", line 1
  File "<stdin>", line 6
```

4.2.5 Protection for data

Two tests were combined into a single class for this particular test. The purpose was to insure that public variables could be written and read by all, and that private variables could only be written and read by the class. As seen, it is a success.

```
>>> class myclass:
...     access foo, bar: public
...     access qux: private
...     def setqux(x):
...             self.qux=x
...     def getqux():
...             return self.qux
...
>>> inst=myclass()
>>> inst.foo=3
>>> inst.foo
3
>>> inst.bar=10
>>> inst.bar
10
>>> inst.qux=22
AccessError: qux
```

```
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>> inst.setqux(22)
>>> inst.qux
AccessError: qux
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>> inst.getqux()
22
```

The second test was to insure that both `write` and `read` access parameters worked correctly. This was done by creating a variable that allowed it to be written privately, but accessed publically:

```
>>> class myclass:
...     access var: public read, private
...     def setvar(x):
...             self.var=x
...
>>> inst=myclass()
>>> inst.var=3
AccessError: var
Stack backtrace (innermost last):
  File "<stdin>", line 1
>>> inst.setvar(3)
>>> inst.var
3
```

The final test was to insure that descendant classes could not access private variables. The test also succeeded:

```
>>> class myclass:
```

```
...     access var: private
...     def setvar(x):
...             self.var=x
...     def getvar():
...             return self.var
...
>>> class subclass(myclass):
...     def writetest():
...             self.var=3
...     def readtest():
...             self.var
...
>>> inst=subclass()
>>> inst.writetest()
AccessError: var
Stack backtrace (innermost last):
  File "<stdin>", line 1
  File "<stdin>", line 3
>>> inst.readtest()
AccessError: var
Stack backtrace (innermost last):
  File "<stdin>", line 1
  File "<stdin>", line 5
```

4.2.6 Class construction in C

This functionality is the most difficult to test, because it requires extensive coding for even the simplest of tests. However, this ability is currently being used in the Alice virtual reality toolkit, and this functionality has been documented to work. [DeL93]

Chapter 5

Conclusion

5.1 Summary

The previous version of Python was restrictive as to the abilities of classes. No kind of data protection was provided, the syntax was awkward, neither constructors nor destructors were supported, and classes could only be written in Python, not in C. The purpose of this thesis was both to redesign Python to allow these features and to implement them, an attempt which was successful in all accounts.

Data protection consists of restricting access to both methods and data; an object can either be public, meaning all can access, or private, meaning access is restricted to the owner of the object. This was implemented for methods by allowing blocks of methods to be declared either `public` or `private`. For data, an `access` command was provided to specify the protection.

Previously, methods were required to have an additional parameter, called `self`. The instance variable was then inserted into the parameter tuple before executing the method. Programmers can also write classes completely in C rather than in Python.

Finally, Python also had not allowed the programmer to specify constructors or destructors; that is, methods which are called upon the creation or deletion of an instance, respectfully. An important change was to allow an `__init__` method as a constructor, and a `__del__` method as a destructor.

5.2 Interpretation

Data protection has resulted in a number of benefits. Not only does it insure that programmers will use the provided abstractions and create subclasses or even new classes if they are not, but program reliability is also increased, since the programmer relies on the abstraction rather than the implementation.

The change in syntax for defining class methods has freed programmers from a previously cumbersome syntax. As a result, programs are less likely to fail, and all Python programs will have a standard name for the instance; previously, the instance could be bound to any variable, not just `self`.

The ability to write classes in C provides not only increased execution speed, but has extended the functions available to a class by allowing access to all of those callable from C. Finally, the ability to specify constructors and destructors provides increased power by allowing the programmer to define variables automatically, without the previous requirement of calling an initialization or destruction method manually.

5.3 Recommendations

The current goal is to integrate the changes I have made with the actual release of Python, thus letting a wider range of programmers benefit from the improved object-oriented model. The task should be rather simple, because the latest release of Python has not included any drastic changes.

There are a number of problems that could be fixed, however. When a Python program is imported (the rough equivalent of C's `#include`), a byte compiled file with the extension of `.pyc` is generated. The next time the file is imported, the `.pyc` file will be loaded instead. Unfortunately, due to the implementation method, protection will not be active when a `.pyc` file is imported. While an easy way to solve this problem is to eliminate the `.pyc` file and thus never encounter it, the correct technique would be to reimplement the method of passing the protection information from the compiler to the interpreter. Instead of storing protection data in variables, the information should be inserted into the actual byte code.

Also, variables are now associated with a particular instance; in other words, an instance could have only one variable named `var`. Unfortunately, this implies that, if a class inherits from two classes with two separate variables called `var`, failure will occur. The method C++ uses to overcome this is to allow an instance to have variables bound at each class, rather than at the instance level. This would allow both variables to co-exist, as long as there wasn't any ambiguity about which one was being accessed.

Both of these changes require a fair amount of coding, with minimal changes to the design. However, care must be taken with the second proposal to insure backwards compatibility. When instance variables are created without an `access` definition, the location must be bound, and the location must allow older Python programs to function correctly.

Glossary

Abstract Data Type A data type in which the following are true: that the representation of both the data and the methods are associated with a single object; and that the methods are the only way of accessing the data. [Seb89, page 323]

Class A series of methods which defines the behavior of all instances of that class.

Constructor A method which is called when an instance is created. In Python, the name of the method is `__init__`.

Data Abstraction The programming philosophy which allows data and functions to be associated together, with the provided functions being the sole method of manipulating the data.

Data Protection The ability of a class or instance to limit the availability of its methods or data objects. This forces programmers utilizing those classes to use the provided abstractions.

Destructor A method which is called when an instance is destroyed. In Python, the name of the method is `__del__`.

Dictionary A Python data type which allows names to be associated with any type of data.

Implementation The term used for the process of writing or modifying a computer program.

Inheritance The ability of a class to selectively redefine the methods of another class.

Instance An object whose behavior is specified by the class of which it is a member. While the methods are specified by the class, data objects are associated with the instance. Thus, there may be multiple instances of a particular class; only the methods are shared.

Method Executable code which specifies an action to take when it is called.

Object A structure which consists of variables and methods which can access the variables. In Python, all methods and variables are implemented as objects.

Object-oriented A programming methodology based on the principle of working with objects that have both data and methods associated with them. Operations are then performed by calling the provided methods.

Private A type of protection in which only the class in which the method or data is defined has access.

Public A lack of protection.

Process Abstraction The programming philosophy which allows the programmer to call functions to accomplish certain tasks, rather than reimplement the task each time.

Tuple A Python data type which consists of a series of objects.

Bibliography

[Coo91] William R. Cook. Object-oriented programming versus abstract data types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, Lecture Notes in Computer Science, pages 151–178, New York, 1991. Springer-Verlag.

[DeL93] Robert DeLine. Alice: A rapid prototyping system for three-dimensional, interactive, graphical environments. Master's thesis, University of Virginia, Department of Computer Science, 1993.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley, Reading, Mass., 1983.

[KA90] Setrag Khoshafian and Razmik Abnous. *Object Orientation: Concepts, Languages, Databases, User Interfaces.* John Wiley and Sons, Inc., New York, 1990.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in* COMMON LISP. Addison-Wesley, Reading, Mass., 1989.

[Mey88] Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[Omo91] Stephen M. Omohundro. The Sather language. Technical report, International Computer Science Institute, Berkeley, CA, June 1991.

[PW91] Lewis J. Pinson and Richard S. Wiener. *Objective-C Object-Oriented Programming Techniques.* Addison-Wesley, Reading, Mass., 1991.

[Seb89] Robert W. Sebesta. *Concepts of Programming Languages.* Benjamin/Cummings, Redwood City, CA, 1989.

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Mass., second edition, 1991.

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. Technical report, Stanford University, Stanford, CA, 1987.

[vR93a] G. van Rossum. Python Library reference. Technical report, CWI, Amsterdam, January 1993.

[vR93b] G. van Rossum. Python reference manual. Technical report, CWI, Amsterdam, January 1993.

[vR93c] G. van Rossum. Python Tutorial. Technical report, CWI, Amsterdam, January 1993. Python is available by FTP from ftp.cwi.nl, in /pub/python.

[KA90] [Kee89] [vR93a] [vR93c]