Cooperative Object-Oriented Programming in Python

A. Chan[1]
[1]Department of Mathematics and Computer Science
Fayetteville State University
North Carolina, 28301

Abstract - *In this article we will investigate the object-oriented features of the Python programming language. Python supports encapsulation with limited information hiding; it has full support in polymorphism and inheritance. Therefore, Python fits the definition of object-oriented programming languages. On the other hand, there are some "add-on" features that are commonly available in many main-stream object-oriented languages but are missing in Python. We will examine these features and, whenever possible, provide "work-around" so the user can enjoy the benefits of these missing features. We call this "cooperative object-oriented programming" because when we "simulate" these features, we often need to advise the users not to trespass into the forbidden zone so that the "work-around" can function as expected.*

Keywords: Python, Object-Oriented, Cooperative.

1 Introduction

Python is often advertised as a multi-paradigm programming language. It supports the imperative, object-oriented, and functional paradigms. But many programmers like to refer it as an object-oriented programming language.

According to Tucker and Noonan [10], a programming language is object-oriented if it supports an encapsulation mechanism with information hiding for defining abstract data types, virtual methods, and inheritance. Based on this definition, Python only barely fits into the object-oriented paradigm. It also lacks many features that are commonly available in other main-stream object-oriented languages. Although this does not hurt its status as an object-oriented language, it will make the language less pleasant to use compared to these main-stream object-oriented languages. In order to simulate these missing features, the programmer may have to depend on documentation and assume the users will follow the advice. Therefore, we came up with the term "cooperative object-oriented programming".

In this article, we first examine the features that qualify Python as an object-oriented programming language and discuss its strengths and weakness. We then look at the common features that appear in other object-oriented languages but are absent in Python. If possible, we will suggest techniques to simulate these features.

In Section 2, we discuss some previous work on "cooperative object-oriented programming". Section 3 presents the object-oriented features that are available in Python. Section 4 discusses the features that are commonly available in object-oriented programming languages but are missing in the Python language. We conclude our discussion in Section 5.

2 Previous Works

We are not the first to use the term "cooperative object-oriented programming." The term was first used by Nascimento and Dollimore [4], who talked about how to have a team of programmers cooperatively and efficiently develop object-oriented software using a pre-defined framework. Damm, Hansen, and Thomsen [2] used the same term to refer to the fact that we usually use the object-oriented paradigm to develop large-scale software, thus requiring a large team of programmers to work cooperatively. Silberztein [8] used this term in a slightly different context, in which he created an architecture that consists a large number of object-oriented components cooperating together to perform a required operation (in his case that is natural language processing).

We, however, use the term differently. In this paper, we use "cooperative object-oriented programming" to refer to the fact that a programming language is lacking some useful features that are usually available in other programming languages. In order to simulate these useful features, sometimes our only option is to document the missing features and ask the users not to touch the forbidden zone. Therefore, when we use the term "cooperative object-oriented programming," we are referring to the necessary cooperation between the users (the programmers) and the library (author).

3 Python's Object-Oriented Features

3.1 Encapsulation with Information Hiding

Encapsulation means putting things together as a single unit. This is not a new concept. Almost all programming languages provide some form of encapsulation, in the form of subroutines (procedures) and functions, which encapsulate the behaviors; and records

(structures) which encapsulate the data. What is unique in object-oriented programming is the encapsulation of behaviors and data into a single unit, which usually appears in the form of a class. We usually call the encapsulated data attributes and the encapsulated behavior methods. When a method is invoked, a message is sent to the object (and the object will respond to the message by executing the behavior encapsulated in the method). There is a distinction between a method and a message. A message is the flowing of execution between the sender (caller) and the receiver (the object), while a method is the implementation of instruction to the system on how to respond to the message.

Information hiding, on the other hand, is a new concept in object-oriented programming. The purpose of information hiding is to solve a common problem – the programmers' abuse of library code. It is a common phenomenon that if the programmers have access to some internal states/methods of a class, eventually some programmers will access those internal states/methods, either intentionally or unintentionally. The abuse makes maintenance of the code much more difficult than necessary. Therefore, it is often necessary to hide the internal working of a class from the users. The mechanism for information hiding is visibility. Most languages provide at least two levels of visibility (e.g. public and private in Smalltalk), some provide more (for example, public, private, protected, and package in Java). Python, based on its design, does not support visibility, and hence no information hiding. The Python standard suggests that names starting with underscores should be treated as private, and therefore users should not touch these names. However, there is no mechanism to actually prevent users from accessing "private" names. Python 3.0 uses name mangling to make accessing these "private" names more complicated, but it is still possible to access these "private" names, although it may require using some less-obvious methods. In fact, the Python Tutorial [7] claims that this "can even be useful in special circumstances, such as in the debugger."

Therefore, information hiding in Python depends on advisement; that is, advising the users not to touch the "private" names; and hoping that the users will follow the advice. This makes Python very suitable for small scale, single-person projects; however, it may be difficult to manage large scale, multi-programmer projects.

3.2 Virtual Methods

Virtual methods (also known as polymorphism) are free in Python. Names in Python do not need to be declared to be of specific types; in fact, the same name can be rebound to different type of values. For example:

```
phone = 5551234     # bound to a numeric value
phone = '555-1234' # now rebound to a string value
```

Therefore, it is impossible for the Python interpreter to know in advance what the actual type of a name is. When a message is sent to an object (usually through a name), the Python interpreter must first find out the following:

1. the object that is bound to the name;
2. whether the object has a method (implementation) for the given message or not; and
3. if yes, let the object respond to the message (execute the appropriate method).

It is easy to notice that the above is exactly how virtual methods work in other languages. In other words, all methods in Python are virtual methods.

We note that Python 3.0 provides a way to allow the methods to check for the types of objects through function annotation [1]. However, the process described above is still the same; and all Python methods are still virtual methods.

3.3 Inheritance

Inheritance is an essential feature of every object-oriented programming language. Without inheritance, a programming language can only be called at most to be object-based, but not object-oriented. However, Python's support for inheritance is a bit non-standard. In most other object-oriented languages, inheritance allows subclasses to inherit all data/behaviors from the parent class(es). The subclasses can then augment, modify, or suppress the inherited data/behaviors. On the other hand, Python subclasses only inherit behaviors, while data are not automatically inherited [3]. Although this will not be a major problem in most cases, sometimes it can introduce subtle bugs.

A Python subclass inherits from its parent class(es) all behaviors (methods implementation). The subclass can then be free to augment, modify and/or suppress these behaviors. Data in Python classes are mostly defined and initialized in the constructor (the `__init__` method). However, unlike other object-oriented programming languages (e.g. Java), which allow the constructor to implicitly invoke the constructor of the superclass, the constructor of a Python subclass will NOT automatically invoke the constructor of the parent class. There is in fact no mechanism to enforce that the constructor of the parent class is invoked at all. If the programmer "forgets" to call the parent class's constructor, then all definitions and initialization of data in the parent class are lost (since data are not inherited automatically), and later, when a method is trying to access these data, errors arise.

Therefore, it is very important to advise the users to remember to invoke the constructor(s) of the parent class(es) in the constructor of the subclass.

We now have looked at the three essential requirements for object-oriented languages and seen that Python more or less supports all these requirements (with some abnormalities). We will next look at some additional "common" features that make object-oriented programming a pleasure and investigate how they can be supported or simulated in Python.

4 Common Non-Object-Oriented Features

4.1 Abstract Methods and Abstract Classes

Abstract methods are those that the programmers only declare the signatures of the methods, but do not provide implementation. This allows programmers to declare the availability of the (abstract) methods without actually implementing them. Abstract methods may be very useful in situations. For example, say we have a `Shape` class as a parent class, and the different specific shapes (e.g. `Circle`, `Polygon`, etc.) as subclasses. It may be useful to declare a `draw` method in the `Shape` class, so we know that this method is available to all subclasses and how to invoke it, but it really makes no sense to implement the `draw` method in the `Shape` class as each specific class will draw itself differently.

Abstract classes are those classes that contain abstract methods, or the programmers specifically want them to be abstract. Abstract classes cannot be instantiated as the implementation is not really completed yet. For example, we cannot (should not) instantiate a `Shape` object as there is no concrete `Shape` – we can have circles, polygons, and so on, but we do not have concrete shapes, so it makes no sense to instantiate a `Shape` object.

In C++, abstract methods are marked with the `virtual` keyword; and any class with virtual methods is abstract class, and hence, cannot be instantiated. In Java, abstract methods are marked with the `abstract` keyword; and any class with abstract methods is abstract class, and hence, cannot be instantiated. Furthermore, Java programmers can also mark a class to be abstract even when it contains no abstract method, hence again, making it impossible to be instantiated. This is sometimes useful. For example, although each shape will be drawn differently, there may be an initialization sequence that is common to all shapes that will want to draw themselves. In this case, we can put the initialization code in the `draw` method of the superclass, and let the subclasses to invoke this superclass method to initialize the drawing.

In Python, there are NO abstract methods, and there are NO abstract classes, at least not up to Python 2.5. Norvig [5] suggested using a non-existing keyword `abstract` to cause exception when the abstract method is invoked. This depends on the fact that no one is defining a variable or function named `abstract`. Furthermore, the exception will be raised only when the abstract method is invoked; there is still no mechanism to prevent an object of a class with abstract methods to be instantiated. One work around is to use the same trick in the constructor (`__init__`) of the abstract class, making the constructor "abstract". This, however, will also prevent the subclasses to invoke this abstract constructor to finish initialization. Therefore, it is not a complete or satisfactory solution.

However, the Python development team is listening. As a result, abstract classes and abstract methods are now supported starting in Python 2.6. To use this new feature, we must import `ABCMeta` class and `abstractmethod` from module `abc`. Then we can mark a method to be abstract using the @abstractmethod decorator. Whenever we instantiate a class with abstract methods, we will get a `TypeError` exception. The following example is taken from the Python Documentation [6]:

```
from abc import ABCMeta, abstractmethod

class Drawable():
    __metaclass__ = ABCMeta

    @abstractmethod
    def draw(self, x, y, scale=1.0):
        pass

    def draw_doubled(self, x, y):
        self.draw(x, y, scale=2.0)

class Square(Drawable):
    def draw(self, x, y, scale):
        ...
```

4.2 Named Constants

Many languages support the idea of named constants. In Fortran, we use the keyword "`PARAMETER`" to mark a name to be constant; in C/C++, the keyword is "`const`"; in Java the keyword is "`final`". In fact, Java extends this idea to methods and classes. In Java, the "`final`" keyword basically means "fixed" and "non-changeable". Therefore, a final variable is an unchangeable name, i.e., a constant; a final method is a method that cannot be overridden; and a final class is a class that cannot be extended (subclassed).

However, there is no notion of constants in Python. (Python has immutable objects, but this is a completely different concept.) Every attribute in a Python class is changeable; every method in a Python class can be overridden; and every Python class can be subclassed.

In an Internet forum [9], there is a very interesting solution to the lack of final variables in Python. The idea is to override the __setattr__ method, which is implicitly invoked when a value is assigned to an attribute, to raise an exception whenever a new value is assigned to an existing attribute:

```
class WriteOnceReadWhenever:
   def __setattr__ (self, attr, value):
      if hasattr (self, attr):
         raise Exception ("Attempting to alter read-only value")

      self.__dict__[attr] = value
```

This actually works, to some extent. The problem is that ALL attributes in this class are now "write once only". This can be fixed by introducing some naming convention so the __setattr__ can check if a name should be a constant or not. For example, we can say that all variables with name beginning with "constant_" should be constants. Then we can modify the code as follows:

```
class WriteOnceReadWhenever:
   def __setattr__ (self, attr, value):
      if attr[:9] == 'constant_' and hasattr (self, attr):
         raise Exception ("Attempting to alter read-only value")

      self.__dict__[attr] = value
```

We note that it is still possible to directly modify self.__dict__ to bypass the checking.

5 Conclusion

We have investigated the object-oriented features of the Python programming language. Python supports encapsulation with limited information hiding; it has full support in polymorphism and inheritance. Therefore, Python fits the definition of object-oriented programming languages. On the other hand, there are many "add-on" features that are commonly available in many main-stream object-oriented languages but are missing in Python. We suspect that this is the main reason why Python is still not popularly used in the industries. However, this does not withhold Python to be an effective and ideal academic language to teach object-oriented programming. In fact, more and more colleges now use Python as their CS0/CS1 language. To efficiently use Python as an object-oriented programming language, we have to largely depend on advisement and documentation. We call this style of programming "Cooperative Object-Oriented Programming" and Python is one of the languages that depend on this style.

6 Acknowledgement

The publication of this paper is partially supported by Fayetteville State University's UCAMCS project which is funded by NSF grant NSF DUE 0631065.

7 References

[1] Alchin, "Pro Python, Advanced Coding Techniques and Tools," Apress, 2010.

[2] Damm, Hansen, and Thomsen, "Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Hague, Netherlands, P518-525, 2000.

[3] Goldwasser and Letscher, "Object-Oriented Programming in Python," Prentice Hall, 2008.

[4] Nascimento and Dollimore, "A Model for Cooperative Object-Oriented Programming," Software Engineering Journal, Volume 8 Issue 1, P41-48, 1992.

[5] Norvig, "The Python IAQ: Infrequently Answered Questions," available at http://norvig.com/python-iaq.html.

[6] Python Documentation Team, "Python Documentation," available at http://www.python.org/doc/.

[7] Rossum, "Python Tutorial," available at http://www.python.org/.

[8] Silberztein, "NOOJ: A Cooperative Object Oriented Architecture for NLP," in Proceedings of the 5th INTEX workshop, Marseille, France, 2002, available at http://www.nooj4nlp.net/NooJArchitecture.pdf.

[9] Stack Overflow Forum, "'final' Keyword Equivalent for Variables in Python?" available at http://stackoverflow.com/questions/802578.

[10] Tucker and Noonan, "Programming Languages: Principles and Paradigms," 2nd edition, McGraw Hill Higher Education, 2007.