Python's object model, with a comparison to Java's object model

Luby Liao
Department of Math/CS
University of San Diego
San Diego, CA 92110
liao@sandiego.edu

ABSTRACT

This paper presents the author's lectures on Python's [1] object model with a comparison to Java's object model in his Programming Languages class.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

HTML, HTMLgen, Object, Java, Python

1. INTRODUCTION

This paper presents lectures I give to my Programming Language class. The goal is to introduce Python's [1] object model with a comparison to Java's object model and see how the differences impact their solutions to some common types of problems. The example problem we are solving here is to write a class library to generate HTML. We call this problem *HTMLgen*.

A typical problem in CS1 deals with bank account classes. In such classes, all instances have the same set and therefore the same number of attributes. This instance uniformity is the basis of Java's object model and is clearly reasonable for this type of problem.

In contrast, consider the *HTMLgen* problem. It would be better if an *Anchor* element that does not use the *target* attribute does not have a *target* instance variable. Two instances of the *Anchor* class can then have different sets of instance variables; one has *target* and *onClick*, while the other does not because it does not use them. HTML elements can potentially have a large number of attributes, but typically only use a handful. For example, a TABLE or a TD element can have about twenty attributes, but usually uses just a few. It is clearly desirable if a language does not require all instances of a class to have the same set of attributes.

The programming language Python offers such an object model. As a result, we can write a Table class in Python for which different Table elements have different sets of attributes. Additionally,

1. Python offers facilities that allow easy definition, instantiation and invocation of methods (including constructors) that have a large number of arguments.

2. Python's object model allows a class to store the default values of the instances' variables in its class attributes. For example, we can define the attribute *border* of the *Table* class to be 0. If a Table instance does not define a border instance variable, then the instance acquires the class attribute's border value. On the other hand, if a Table instance wishes to use a non-zero border, it has only to define its own border instance variable to override the corresponding class attribute.

It will become apparent that Python's object model and its convenient features of function definition and invocation contribute to program writability and readability. This empowers students to write maintainable code and can effectively engage them to work on real projects. An additional challenge to the students is to solve the HTMLgen problem in Java.

Robin Friedrich wrote an `HTMLgen.py` Python module [2] that solved the *HTMLgen* problem beautifully. For example, using his `HTMLgen.Table` class, we can quickly generate web tables without writing HTML.

```
[liao@zen htmlgen]$ python2.3
Python 2.3.2 (#1, Oct  6 2003, 10:07:16)
>>> from HTMLgen import Table
>>> table = Table()
>>> table.heading =['First', 'Second', 'Third']
>>> table.body = [[1,2,3],
...               [4,5,6],
...               [7,8,9]]
>>> print table
<TABLE border=2 cellpadding=4 cellspacing=1
   width="100%">
```

```
 <TR Align=center> <TH ColSpan=1>First</TH>
  <TH ColSpan=1>Second</TH>
  <TH ColSpan=1>Third</TH></TR>
 <TR><TD Align=left >1</TD>
  <TD Align=left >2</TD>
  <TD Align=left >3</TD>
 </TR>
 <TR><TD Align=left >4</TD>
  <TD Align=left >5</TD>
  <TD Align=left >6</TD>
 </TR>
 <TR><TD Align=left >7</TD>
  <TD Align=left >8</TD>
  <TD Align=left >9</TD>
 </TR>
</TABLE><P>
```

This is rendered as:

First	Second	Third
1	2	3
4	5	6
7	8	9

My frequent application of this is to use the result set of an SQL query as the body of a `HTMLgen.Table` for quick publication to the Web.

This paper is modeled after chapter seven of Aaron Watter's book [3]. I present six progressively better Python classes to generate external web links.

2. A STRAIGHTFORWARD VERSION

Following `HTMLgen.py`, I call my *simplified* class for external web links `Href`. In the first version, all constructor parameters are required. Readers unfamiliar with Python should note that, unlike Java, in a method, a reference to the instance is explicitly the first parameter (see line 3), and is named *self* by convention. Each parameter (lines 4-9) except the first corresponds to a potential attribute for the `A` tag. The constructor `__init__` simply uses its arguments to establish the state of an instance. The `__str__` method is the Python analog of Java's `toString`. It produces a string representation of a `Href` object, which is the main purpose of the `Href` class. The `__str__` (lines 23-36) is copied verbatim from `HTMLgen.Href`. The code for `__str__` method will not be duplicated elsewhere in this paper since it remains unchanged throughout. The class comes with self-testing code enclosed in an `if` block (lines 39-47). This is effectively the same as the `main` class method of a Java class.

```
 1 class Href:
 2
 3   def __init__(self,
 4                url,
 5                text,
 6                target,
 7                onClick,
 8                onMouseOver,
 9                onMouseOut):
10     """ A straightforward constructor.
11         All parameters are required.
12         The constructor simply use the arguments
13         to establish the state of the object
14     """
15
16     self.url = url
17     self.text = text
18     self.target = target
19     self.onClick = onClick
20     self.onMouseOver = onMouseOver
21     self.onMouseOut = onMouseOut
22
23   def __str__(self):
24     s = ['<A HREF="%s"' % self.url]
25     if self.target:
26       s.append(' TARGET="%s"' % self.target)
27     if self.onClick:
28       s.append(' onClick="%s"' % self.onClick)
29     if self.onMouseOver:
30       s.append(' onMouseOver="%s"' %
31                self.onMouseOver)
32     if self.onMouseOut:
33       s.append(' onMouseOut="%s"' %
34                self.onMouseOut)
35     s.append('>%s</A>' % self.text)
36     return ''.join(s)
37
38
39 if __name__ == '__main__':
40   print Href('http://www.sandiego.edu',
41              'University of San Diego',
42              '_blank', None, None, None)
43   print Href('http://www.sandiego.edu',
44         'University of San Diego',
45         None, None,
46         "document.usd.src='images/usd.jpg'",
47         "document.usd.src='images/USD.jpg'")
```

The output of the testing code (lines 40-47) is

```
<A HREF="http://www.sandiego.edu" TARGET="_blank">
   University of San Diego
</A>
<A HREF="http://www.sandiego.edu"
  onMouseOver="document.usd.src='images/usd.jpg'"
  onMouseOut="document.usd.src='images/USD.jpg'">
    University of San Diego
</A>
```

This initial version has obvious drawbacks. As noted, all arguments are required even if the corresponding attributes are not used in the `A` tag. For example, even though the second link (lines 43-47) does not use the `target` attribute, a trivial `target` argument must still be provided. In addition, the only way to provide arguments is by matching the parameters by position. This becomes difficult and error-prone when the number of parameters is numerous.

3. EASING INSTANTIATION

The next version solves both problems by using Python's default values for parameters (lines 5-10) and its support for both positional (lines 26-27) and keyword arguments (lines 28, 33-36) for function invocation. The default value for a parameter can be provided when declaring a function (lines 5-10). Subsequently such argument needs not be provided

in a function call (e.g., line 23). In that case, default values are used.

This version, named `Href_2.py` on line 1, differs from the first in two places:

- The signature of the constructor (lines 5-10)
- The constructor call (lines 23, 26-28, 31-36) omits the arguments it does not really use

```
 1 # Href_2.py
 2 class Href:
 3
 4   def __init__(self,
 5                url='',
 6                text='',
 7                target = None,
 8                onClick = None,
 9                onMouseOver = None,
10                onMouseOut = None):
11
12     self.url = url
13     self.text = text
14     self.target = target
15     self.onClick = onClick
16     self.onMouseOver = onMouseOver
17     self.onMouseOut = onMouseOut
18
19   # def __str__(self): <is omitted from now on>
20
21 if __name__ == '__main__':
22
23   a = Href()
24   print a
25
26   b = Href('http://www.sandiego.edu',
27            'University of San diego',
28            target='_blank')
29   print b
30
31   c = Href('http://www.sandiego.edu',
32         'University of San diego',
33         onMouseOver=
34           "document.usd.src='images/usd.jpg'",
35         onMouseOut=
36           "document.usd.src='images/USD.jpg'")
37   print c
```

The testing code (lines 23-37) produces the following output:

```
<A HREF=""></A>
<A HREF="http://www.sandiego.edu" TARGET="_blank">
   University of San Diego</A>
<A HREF="http://www.sandiego.edu"
  onMouseOver="document.usd.src='images/usd.jpg'"
  onMouseOut="document.usd.src='images/USD.jpg'">
    University of San Diego</A>
```

4. ELIMINATING USELESS VARIABLES

A problem with `Href_2.py` is that all instances of `Href` use the same number of instance variables, even when these variables do not contribute to the state of an instance, such as when `target` equals `None`. A solution is to move the potentially useless names from the namespace of the instances (lines 14-17 of `Href_2.py`) to the namespace of the class (lines 4-7) and use conditionals (lines 19-26) to add names to the instance's namespace only when non-trivial arguments are specified. This works because of the rich semantics of name resolution for an instance: name resolution starts with the namespace of the instance and, if unresolved, continues on to the namespace of the class.

```
 1 # Href_3.py
 2 class Href:
 3
 4   target = None
 5   onClick = None
 6   onMouseOver = None
 7   onMouseOut = None
 8
 9   def __init__(self,
10                url='',
11                text='',
12                target = None,
13                onClick = None,
14                onMouseOver = None,
15                onMouseOut = None):
16
17     self.url = url
18     self.text = text
19     if target:
20       self.target = target
21     if onClick:
22       self.onClick = onClick
23     if onMouseOver:
24       self.onMouseOver = onMouseOver
25     if onMouseOut:
26       self.onMouseOut = onMouseOut
```

The testing code is omitted here as is its output, which are identical to those of `Href_2.py`. The high point of this version is that we factor the potentially useless instance attributes into the class's attributes. As a result, instances will share these class attributes when these attributes do not contribute to their states. Instances that really do use the attribute *target* will define an instance variable `target`, shadowing the class's attribute with the same name. Instances not using the target attribute will not have such attribute. As a result, different instances will have different numbers and sets of instance attributes. This is quite different from the static nature of Java class instances: all of them have the same set of attributes at runtime.

The following shows indeed that the Href instances `a, b,` and `c` all have different set and number of instance variables:

```
[liao@zen papers]$ python2.3
Python 2.3.2 (#1, Oct  6 2003, 10:07:16)
>>> from Href_3 import Href
>>> a = Href()
>>> a.__dict__
{'url': '', 'text': ''}
>>> b = Href('http://www.sandiego.edu',
...   'University of San Diego',
...   target='_blank')
>>> b.__dict__
```

```
{'url': 'http://www.sandiego.edu',
 'text': 'University of San Diego',
 'target': '_blank'}
>>> c = Href('http://www.sandiego.edu',
...   'University of San Diego',
...    onMouseOver=
...      "document.usd.src='usd.jpg'",
...    onMouseOut=
...      "document.usd.src='USD.jpg'")
>>> c.__dict__
{'url': 'http://www.sandiego.edu',
 'text': 'University of San Diego',
 'onMouseOver': "document.usd.src='usd.jpg'",
 'onMouseOut': "document.usd.src='USD.jpg'"}
>>>
```

It is not implied that Java's object storage is less efficient than Python's. In fact, each Python class instance needs to store its own attribute names, whereas Java instances need only to store the attribute values using indexed displacements, owing to the fact that all instances of the same class have the same set of attributes. It is also clear that factoring an instance attribute into its class in Python will further slow down attribute access (since time is required to follow pointers).

5. SIMPLICITY

How can we avoid having to duplicate essentially the same if statements four times in lines 19-26 of `Href_3.py`? We can achieve this with a loop (line 17-18) by using the extra keyword arguments that are signified by the syntax `**kw` (line 12), where two asterisks are required but `kw` is an arbitrary identifier.

```
 1 # Href_4.py
 2 class Href:
 3
 4   target = None
 5   onClick = None
 6   onMouseOver = None
 7   onMouseOut = None
 8
 9   def __init__(self,
10                url='',
11                text='',
12                **kw):
13
14     self.url = url
15     self.text = text
16
17     for item in kw.keys():
18       self.__dict__[item] = kw[item]
```

With the declaration of `**kw` on line 12, a call can tag on any number of extra keyword/value pairs at the end of the argument list. For example,

```
Href('http://www.sandiego.edu', 'USD',
  onMouseOver="document.usd.src='images/usd.jpg'",
  target='_top')
```

is legal. Python runtime will collect all extra keyword arguments provided in the call in a dictionary named `kw`, or whatever identifier the definition of `__init__` uses. The constructor then is able to use the variable `kw` in any way it wishes. In our case, the constructor iterates through `kw`'s keys and creates instance variables for these keys with the given values. In short, it uses a simple loop of two lines to establish the instance's state, rather than a block of statements whose size is linear in the number of attributes.

6. DISALLOWING MEANINGLESS ARGUMENTS

A problem with the last version is that users can provide any key/value pair to the constructor. For example, this would be a legitimate call:

```
print Href(age=3, sex='m', degree='ph.d.')
```

even though none of the arguments make sense to the constructor whose purpose is to build a web link instance.

In reality, a constructor call should be allowed only a finite set of meaningful keyword arguments. A solution is:

- first populate the namespaces of the class and the instances with meaningful names, as our code has already done: the class defines four names in its namespace (lines 4-7), and the constructor (lines 14-15) defines two names in an instance's namespace.
- before accepting an argument, we check in line 20 with `getattr` that it is already in the namespace of either the instance or the class. Otherwise the argument is deemed meaningless, and the constructor raises an exception.

To understand `getattr` (line 20), readers are again reminded that resolution of a name for an instance starts with the namespace of the instance. If it is not found there, the resolution is delegated to the class's namespace. `Getattr` will raise an exception if the name cannot be resolved.

```
 1 # Href_5.py
 2 class Href:
 3
 4   target = None
 5   onClick = None
 6   onMouseOver = None
 7   onMouseOut = None
 8
 9   def __init__(self,
10                url='',
11                text='',
12                **kw):
13
14     self.url = url
15     self.text = text
16
17     for item in kw.keys():
18       # see if self  knows about item
19       try:
20         getattr(self, item)
21         self.__dict__[item] = kw[item]
22       except:
23         raise KeyError("%s is an illegal \
24                     keyword argument!" % `item`)
25
```

```
26
27 if __name__ == '__main__':
28
29   print Href(age=3, sex='m', degree='ph.d.')
```

The constructor call (line 29) with meaningless arguments raised an exception like this:

```
Traceback (most recent call last):
  File "<stdin>", line 37, in ?
  File "<stdin>", line 22, in __init__
KeyError: 'sex' is an illegal keyword argument!
```

7. FACTORING GENERIC CODE

Note that lines 17 to 24 of the `Href_5.py`'s constructor are totally generic. It simply checks to see if a keyword argument provided is known to the instance. If it is, then the name/value pair is added to the namespace of the instance; otherwise an exception is raised. We should factor this code into a superclass `HTMLElement` so it can be shared among other HTML elements, such as Table, Form, etc.

```
 1 # Href_6.py
 2 class HTMLElement:
 3
 4   def __init__(self, **kw):
 5
 6     for item in kw.keys():
 7       # see if self  knows about item
 8       try:
 9         getattr(self, item)
10         self.__dict__[item] = kw[item]
11       except:
12         raise KeyError("Unsupported argument!")
13
14 class Href(HTMLElement):
15
16   target = None
17   onClick = None
18   onMouseOver = None
19   onMouseOut = None
20
21   def __init__(self,
22                url='',
23                text='',
24                **kw):
25
26     self.url = url
27     self.text = text
28     HTMLElement.__init__(self, **kw)
29
30 if __name__ == '__main__':
31
32   print Href('http://www.sandiego.edu',
33              'University of San Diego',
34              target='_blank')
35   print Href('http://www.sandiego.edu',
36         'University of San Diego',
37         target='_blank',
38         onMouseOver="document.usd.src='usd.jpg'",
39         onMouseOut="document.usd.src='USD.jpg'")
```

Line 14 declares `Href` as a subclass of `HTMLElement`. In line 28, subclass's constructor calls the factored out method in the superclass.

8. CONCLUDING COMMENTS

Pedagogically, the *HTMLgen* problem is not only intellectually exciting, it is very motivating, because it is perceived by students as practical. The `HTMLgen.py` class library is well written and easy to read. It is a significant piece of open source software that students can benefit not only from using, but also from studying critically.

This paper shows that Java's and Python's underlying object models are very different; it is useful for students to compare these differences and see how they impact their solutions to problems such as *HTMLgen*.

There are other differences between Java's and Python's object orientation styles. Some examples include:

- In both languages the first argument of an instance method must be a reference to the instance, in Java this argument is implicit and must not be declared; in Python it must be explicitly declared. This was briefly mentioned in Section 2.
- Java forces constructor chaining (often implicitly), meaning that a constructor call is guaranteed to call the constructors of all superclasses. In contrast, if a Python constructor wishes to call its ancestors' constructors, it must do so explicitly. This was demonstrated in Section 7. To wit, in line 28, `Href`'s `__init__` needs to call `HTMLElement`'s `__init__` explicitly. The comparison of the implicit versus explicit instance reference and constructor chaining deepens students' understanding of such concepts.
- The above example also shows that a Python constructor is just a method (line 28), while in constrast, a Java constructor is not.
- Java allows a class to subclass from only one class and any number of interfaces; a Python class can subclass from any number of classes. Python's multiple inheritance is easy to understand and use.
- Finally, a Python class instance can dynamically add or delete a method, instance variable or superclass; Java cannot.

It is beyond this paper's scope to make an exhaustive comparison.

9. REFERENCES

[1] http://python.org/.

[2] R. Friedrich. http://starship.python.net/lib.html.

[3] A. Watters, G. V. Rossum, and J. C. Ahlstrom. *Internet Programming with Python*. Hungry Minds, 1996.