

**CSC 520 Fall 2010, Object Closures and Metaclasses in Python, Dec. 2, Dr. Dale Parson**  
**Also an overview of Object-Oriented Databases**

```
>>> def addcon(con):           # This is a Python functional closure.
...     def helpadd(var):
...         return con + var
...     return helpadd
...
>>> f = addcon(3)
>>>
>>> f
<function helpadd at 0x00D960F0>
>>> f(2)
5
>>> f(7)
10
```

```
>>> class addcon(object): # This Python class provides an equivalent "object closure." This could be C++ or Java.
...     def __init__(self, con):
...         self.con = con
...     def eval(self, var):
...         return self.con + var
...
>>> obj = addcon(3)
>>> obj
<__main__.addcon object at 0x00D93D50>
>>> obj.eval(2)
5
```

```
>>> unboundMethod = addcon.eval # Bind variable unboundMethod to a method unbound to an object.
>>> unboundMethod
<unbound method addcon.eval> # The C++ equivalent is a pointer to a member function (strange syntax).
>>> unboundMethod(obj, 2) # Unbound method requires the object as the first argument. No Java equivalent.
5
```

```
>>> boundMethod = obj.eval # Bound method binds both object and method. Use the same as a reference to a
function.
>>> boundMethod
<bound method addcon.eval of <__main__.addcon object at 0x00D93D50>>
>>> boundMethod(2) # No counterpart in C++ or Java.
5
```

```

>>> def addcon(con): # A Python class definition gets the function closure of its surrounding function.
...     class helpadd(object):
...         def eval(self, var):
...             return con + var # Note that self.con is not used here. Con is in the outer function's environment.
...     return helpadd().eval
...
>>> f = addcon(3)
>>> f
<bound method helpadd.eval of <__main__.helpadd object at 0x00D93F10>>
>>> f(2)
5
>>> f(7)
10

```

~parson/ProcLang/objects\_minimake/mmcompile.py from Spring 2010 uses object closures as op codes for a mini-make virtual machine.

```

def compileRules(symboltable):
    # Using my solution to Assignment 3 in ~parson/ProcLang/compile_minimake
    # file mmcompile.py as a model for the algorithms, re-implement function
    # compileRules as follows.
    # 1. There are *NO* nested functions compileDependency, opcodeDependency,
    #    compileTimeCheck, opcodeTimeCheck, compileAction or opcodeAction.
    # 2. Replace the above functions with three Python classes NESTED
    #    WITHIN THIS FUNCTION named classDependency, classTimeCheck and
    #    classAction. Each is derived from Python's "object" base class.
    # 3. Each of these classes has a constructor (named __init__ according
    #    to Python built-in convention) that takes a "self" parameter
    #    for the "this" pointer, followed by the parameters that will be
    #    needed to run the opcode later. These additional parameters are
    #    simply the parameters accepted by compileDependency,
    #    compileTimeCheck and compileAction of Assignment 3, along with
    #    any other variables or parameters bound by the opcode closures
    #    of Assignment 3. Instead of storing these parameters in a
    #    closure, the __init__ constructor must store them in fields in
    #    the (classDependency, classTimeCheck or classAction) object being
    #    constructed.
    # 4. Each of these classes must define a method called "eval" that
    #    takes the object reference (self) and a timestamp as parameters,
    #    and that returns a timestamp according to the logic of the opcode
    #    closures of Assignment 3.
    # 5. In addition to defining these three nested classes, function
    #    compileRules must initialize the result dictionary as before,
    #    and then populate it with a mapping from executable target names
    #    to opcodes as before. The compilation code of compileRules uses
    #    the same basic logic as Assignment 3, but for Assignment 5 an
    #    opcode is a reference to a METHOD BOUND TO AN OBJECT. That means
    #    that, for every a) dependency, b) timecheck and c) action, compileRules
    #    must construct an object of class classDependency, classTimeCheck
    #    or classAction using the appropriate parameters, and then must store
    #    a pointer to the "eval" method FOR THAT OBJECT. I will hand out
    #    some demo code concerning Python classes, objects, unbound method
    #    references and bound method references that will clarify the

```

```

# form that an opcode must take.
#
# SUMMARY: This assignment replaces the function closures of Assignment 3
# with object closures as defined in Section 3.6.3 of the textbook.
#
# SEE: ~parson/ProcLang/compile_minimake/mmcompile.py for the algorithms
# for compileRules and the opcodes.
#
class classDependency(object):
    # Opcode class for checking a make dependency.
    def __init__(self, opcodeTable, dependency):
        self.opcodeTable = opcodeTable
        self.dependency = dependency
    def eval(self, timestamp):
        mtime = 0
        isok = False
        if (self.opcodeTable.has_key(self.dependency)):
            isok = True # It is a rule.
            executeRules(self.opcodeTable, self.dependency)
        try:
            mtime = os.stat(self.dependency).st_mtime # time of last mod.
            isok = True # It is a file.
        except Exception:
            mtime = timestamp + 1 # No file, assures actions will fire.
        if (not isok):
            raise ValueError, \
                self.dependency + " is not a file and is not a target."
        return mtime
class classTimeCheck(object):
    # Opcode class for comparing target timestamp to dependencies'.
    def __init__(self, target):
        self.target = target
    def eval(self, timestamp):
        mtime = 0
        try:
            mtime = os.stat(self.target).st_mtime # time of last mod.
        except Exception:
            mtime = 0 # No target file, assure that it will fire.
        return timestamp if timestamp > mtime else 0
class classAction(object):
    def __init__(self, act):
        self.act = act
    def eval(self, timestamp):
        sys.stdout.write(self.act + "\n");
        sys.stdout.flush() # Make sure it appears before any explosion.
        processStatus = os.system(self.act)
        if (processStatus):
            raise ValueError, "error on: " + self.act + \
                ", exit status = " + str(processStatus)
        return timestamp if timestamp >= 1 else 1 # Keep VM running.
# Main code of compileRules.
result = {}
for t in symboltable.keys():
    deps, acts = symboltable[t]

```

```

ops = []
if (deps):
    for dep in deps:
        obj = classDependency(result, dep)
        ops.append(obj.eval)
        obj = classTimeCheck(t)
        ops.append(obj.eval)
    for act in acts:
        obj = classAction(act)
        ops.append(obj.eval)
result[t] = ops
return result

```

```

def executeRules(opdictionary, target):

```

```

    """

```

```

    This function accepts as input parameters an opcode dictionary
    returned by compileRules and a rule target name that is a key in
    that dictionary. It executes the compiled opcodes for target and
    all rules on which it depends, invoking action lists in
    the operating system shell for all triggered rules.

```

```

    """

```

```

    if (not opdictionary.has_key(target)):

```

```

        raise ValueError, "Rule target " + target + " is unknown."

```

```

    ops = opdictionary[target]

```

```

    timestamp = 0

```

```

    try:

```

```

        timestamp = os.stat(target).st_mtime # time of last mod.

```

```

    except Exception:

```

```

        timestamp = 0 # No target file, assure that it will fire.

```

```

    for op in ops:

```

```

        newstamp = op(timestamp)

```

```

        if (not newstamp):

```

```

            break

```

```

        if (newstamp > timestamp):

```

```

            timestamp = newstamp

```

Python metaclasses allow programmers to customize class-like compile-time mechanisms.

~parson/ThryLang/metaclasses/metasnoop.py

```
1 # metasnoop.py -- D. Parson, CSC 580, Spring, 2009
2
3 from sys import stdout
4
5 # A quick look at what a meta-class can see.
6
7 # 1. Define a meta-class that acts like a regular Python class
8 # except that it tells us what's available for manipulation in
9 # a custom way. The structure of this comes right out of the
10 # PY book Chapter 7 with tweaks.
11
12 class snoop_metaclass(type):
13     def __init__(self, name, bases, dict):
14         stdout.write("DEBUG metaclass PARAM name: " + str(name) + "\n")
15         stdout.write("DEBUG metaclass PARAM bases: " + str(bases) + "\n")
16         stdout.write("DEBUG metaclass PARAM dict: " + str(dict) + "\n")
17         # The next step actually constructs a class.
18         # Note that we could have manipulated dict or in fact any of
19         # these parameters to change the definition of the class
20
21         # Add code here to check whether dict has a __patterns__
22         # list conforming to the __patterns__ list in class "rule1" below,
23         # and if it does, replace that list with the definition of a function
24         # called "test_patterns(self)" that performs those tests on fields in the
25         # object and returns True if the object passes the tests else False.
26         # If there is no "__patterns__" list, generate a function "test_patterns(self)"
27         # that returns a class to the base class test_patterns.
28         # You must use the "exec" operation as illustrated
29         # in the PY Chapter 7 example code to create method test_patterns. The
30         # global environment for this "exec" call is globals() and the
31         # local environment is this dict of this class.
32         # You must then use "setattr()" as documented in the PY book to
33         # attach the "test_patterns" method to the clas being generated by the
34         # meta-class.
35         test_patterns = """def test_patterns(self):
36             return("""
37             if (dict.has_key("__patterns__")):
38                 pats = dict["__patterns__"]
39                 isfirst = True
40                 for atest in pats:
41                     if (isfirst):
42                         isfirst = False
```

```

43         else:
44             test_patterns = test_patterns + " and "
45             test_patterns = test_patterns + atest
46     else:
47         test_patterns = test_patterns + "super(" + name + ",self).test_patterns()"
48         test_patterns = test_patterns + ")\n"
49         print "DEBUG BODY OF TESTER:\n" + test_patterns
50         exec test_patterns in globals(), dict
51         stdout.write("DEBUG POST metaclass PARAM dict: " + str(dict) + "\n")
52         type.__init__(self, name, bases, dict)
53         print "DEBUG AFTER init", str(dir(self))
54         setattr(self, "test_patterns", dict["test_patterns"])
55
56 # 2. Initial demo of a class that defines some relational rules.
57 # The meta-class must create method "test_patterns(self)" that tests whether each new
58 # rule object has fields that satisfy these constraints.
59 # This is a first step in showing how to compile a CLIPS-like pattern
60 # matcher using Python meta-classes.
61 class rule1(object):
62     # Test rules go into class static variable called "__patterns__".
63     __patterns__ = ["self.field1 < 2", "self.field2 > self.field1"]
64     __metaclass__ = snoop_metaclass
65
66 class test_rule1(rule1):
67     def __init__(self, f1, f2):
68         self.field1 = f1
69         self.field2 = f2
70
71 obj1 = test_rule1(-1, 5)
72 print "obj1 tests " + str(obj1.test_patterns())
73 obj2 = test_rule1(5, -1)
74 print "obj2 tests " + str(obj2.test_patterns())

```

## Java and Object-Oriented Databases

**Java Database Connectivity (JDBC)** API provides an interface for connecting to **Relational / SQL databases**.

<http://download.oracle.com/javase/6/docs/technotes/guides/jdbc/index.html>

It is not an object-oriented database API. It does use Java interfaces and classes to connect to a relational DBMS.

Java Data Objects (JDO) is an API for object-oriented database implementations.

<http://db.apache.org/jdo/>

<http://www.oracle.com/technetwork/java/jdo-137135.html>

With JDO you can develop plain old Java objects (POJOs) that are persistent, similar to serialization of object graphs built into `java.io.Serializable`, with numerous added benefits.

- A class can be **Persistence Capable** (a persistence class for the database), **Persistence Aware** (manages database interactions), or Normal (neither of the above).
- **Dirty Field Detection** is automated.
- **Query** applies to combinations of object fields.
- **Transaction** includes `begin()`, `commit()` and `rollback()`.
- **PersistenceManagerFactory** provides class-loader initialization of a commercial or open source datastore implementation. A **PersistenceManager** manages run time interactions thereafter.
- A **Persistence Capable** class must be **Enhanced** by the **Enhancer**, which reads additional meta-data about the Persistence Capable class in either XML meta-data files (JDO 1.0 and 2.0) or Java 1.5 annotation classes (added in JDO 2.1), and then generates Enhanced class counterparts.