Introduction to Jython, Part 2: Programming essentials

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. About this tutorial 2
2. Object-oriented programming in Jython 5
3. Advanced object-oriented programming 13
4. Debugging Jython 25
5. Java support in Jython 29
6. Java thread support in Jython 34
7. Interfacing with Java services 40
8. Jython string processing 46
9. Processing regular expressions 54
10. File I/O in Jython 58
11. A simple Swing GUI 67
12. Wrap-up and resourcesFeedback 73
13. Appendices 76

Section 1. About this tutorial

What is this tutorial about?

This is the second installment in a two-part tutorial designed to introduce you to the Jython scripting language. Jython is an implementation of Python that has been seamlessly integrated with the Java platform. Python is a powerful object-oriented scripting language used primarily in UNIX environments.

In *Part 1* of this tutorial, you learned the basics of Jython, including installation and setup, access options and file compilation, syntax and data types, program structure, procedural statements, and functions. In Part 2 we will delve into some of the more advanced aspects of working with this powerful scripting language, starting with an in-depth introduction to object-oriented programming with Jython. We'll also discuss topics essential to the mechanics of application development in any language, including debugging, string processing, and file I/O.

By the time you have completed this second half of the two-part introduction to Jython, you will be able to write and implement complete functions, classes, and programs in Jython.

Should I take this tutorial?

This tutorial is designed as a progressive introduction to Jython. If you have not completed *Part 1* of the tutorial, you should do so before proceeding to Part 2. Both the conceptual discussion and many of the code examples presented here will be difficult to follow without reference to Part 1.

In this second half of the tutorial,we will cover the following aspects of scripting with Jython:

- Object-oriented programming with Jython
- Debugging
- Java support
- String processing
- File I/O
- Building a Swing GUI application in Jython

To benefit from the discussion, you should be familiar with at least one procedural programming language and the basic concepts of computer programming, including command-line processing. To fully utilize Jython's features you should also be familiar with the basic concepts of object-oriented programming. To fully understand the GUI application example at the end of the tutorial you should have prior experience with Swing GUI programming, although you will be able to glean a lot from the preceding discussion and examples. It will also be helpful to have a working knowledge of the Java platform, because Jython runs on a JVM; although this is not a requirement of the tutorial.

Note that this tutorial is oriented towards Windows systems. All command examples will employ Windows syntax. In most cases similar commands perform the same functions on UNIX systems, although these commands will not be demonstrated.

Tools, code, and installation requirements

You must have Jython 2.1 or higher installed on your development system to complete this tutorial. Your development system may be any ASCII text editor (such as Windows Notepad) combined with the command prompt. The tutorial includes detailed instructions for getting and installing Jython on your system.

To use Jython you must also have a Java Runtime Environment (JRE) installed on your system. It is recommended that you use the latest JRE available (1.4.2 at the time of writing), but any version at or beyond Java 1.2 should work fine. If you are going to use Jython from a browser (that is, as an applet), you must have at least a JRE 1.1 available to the browser. See the Resources on page 73 section to download the latest version of the JDK.

All code examples in this tutorial have been tested on Jython running on the Sun Java 1.4.1 JRE on Windows 2000. Examples should work without change on any similar configuration on other operating systems.

Included with the tutorial is a set of appendices detailing all of the code examples you will use to learn about Jython. All code examples have been tested on Jython running on the Sun Java 1.4.1 JRE on Windows 2000. Examples should work without change on any similar configuration on other operating systems.

About the author

Dr. Barry Feigenbaum is a member of the IBM *Worldwide Accessibility Center*, where he is part of team that helps IBM make its own products accessible to people with disabilities. Dr. Feigenbaum has published several books and articles, holds several patents, and has spoken at industry conferences such as JavaOne. He serves as an Adjunct Assistant Professor of Computer Science at the University of Texas, Austin.

Dr. Feigenbaum has more than 10 years of experience using object-oriented languages like C++, Smalltalk, the Java programming language, and Jython. He uses the Java language and Jython frequently in his work. Dr. Feigenbaum is a Sun Certified Java Programmer, Developer, and Architect. You can reach him at *feigenba@us.ibm.com*.

Acknowledgements

I would like to acknowledge Mike Squillace and Roy Feigel for their excellent technical reviews of this tutorial.

Section 2. Object-oriented programming in Jython

A conceptual overview

Object-oriented programming (OOP) represents the state-of-the-art in software programming technique. OOP is based on the notion of creating a *model* (or simulation) of the target problem in your programs. Properly using OOP techniques reduces programming errors, speeds up software development, and facilitates the reuse of existing code. Jython fully supports the concepts and practice of OOP.

In the following sections I will introduce OOP and describe how it is achieved in Jython. In the next section I will discuss some of the more advanced features of object-oriented programming in Jython.

Objects in Jython

Jython is an *object-oriented* language that completely supports object-oriented programming. Objects defined by Jython have the following features:

- **Identity**: Each object must be distinct and this must be testable. Jython supports the `is` and `is not` tests for this purpose.
- **State**: Each object must be able to store state. Jython provides *attributes* (a.k.a. *fields* or *instance variables*) for this purpose.
- **Behavior**: Each object must be able to manipulate its state. Jython provides *methods* for this purpose.

Note that the `id(object)` built-in function returns a unique integer identity value. So, the expression `x is y` is equivalent to `id(x) == id(y)`.

OOP support in Jython

In its support for object-oriented programming, Jython includes the following features:

- **Class-based object creation**: Jython *classes* are templates for the creation of objects. *Objects* are data structures with associated behavior.

- **Inheritance with polymorphism**: Jython supports *single-* and *multiple-inheritance* . All Jython instance methods are *polymorphic* (or *virtual*) and may be overridden by subclasses.

- **Encapsulation with data hiding**: Jython allows (but does not require) attributes to be hidden, thus permitting access outside the class itself only through methods of the class. Classes implement functions (called methods) to modify the data.

Defining a class

Defining a class is a lot like defining a module in that both variables and functions can be defined. Unlike the Java language, Jython allows the definition of any number of public classes per source file (or module). Thus, a module in Jython is much like a package in the Java language.

We use the `class` statement to define classes in Jython. The `class` statement has the following form:

```
class name ( superclasses ):  statement

  -- or --

class name ( superclasses ):
    assignment
     :

    function
     :
```

When you define a class, you have the option to provide zero or more *assignment* statements. These create class attributes that are shared by all instances of the class. You can also provide zero or more *function* definitions. These create methods. The superclasses list is optional. We'll discuss superclasses a little later in the tutorial.

The class name should be unique in the same scope (module, function, or class). The class name is really a variable bound to the class body (similar to any other assignment). In fact, you can define multiple variables to reference the same class.

Creating a class instance

Classes are used to hold class (or shared) attributes or to create class instances. To create an instance of a class you call the class as if it were a function. There is no need to use a *new* operator like in C++ or the Java language. For example, with the class

```
class MyClass:
    pass
```

the following statement creates an instance:

```
x = MyClass()
```

Adding attributes to a class instance

In Jython (unlike in the Java language) clients can add *fields* (also known as *attributes*) to an instance. Only the one instance is changed. To add fields to an instance (*x*) just set new values on that instance, as shown below:

```
x.attr1 = 1
x.attr2 = 2
  :
x.attrN = n
```

Defining class attributes and methods

Any variable bound in a class is a *class attribute* (or variable). Any function defined within a class is a *method*. Methods receive an instance of the class, conventionally called `self`, as the first (perhaps only) argument. For example, to define some class attributes and methods, you might enter:

```
class MyClass:
    attr1 = 10        # class attributes
    attr2 = "hello"

    def method1(self):
        print MyClass.attr1   # reference the class attribute

    def method2(self, p1, p2):
        print MyClass.attr2   # reference the class attribute

    def method3(self, text):
        self.text = text    # instance attribute
        print text, self.text    # print my argument and my attribute
```

```
    method4 = method3       # make an alias for method3
```

Note that inside a class, you should qualify all references to class attributes with the class name (for example, `MyClass.attr1`) and all references to instance attributes with the `self` variable (for example, `self.text`). Outside the class, you should qualify all references to class attributes with the class name (for example, `MyClass.attr1`) or an instance (for example, `x.attr1`) and all references to instance attributes with an instance (for example, `x.text`, where *x* is an instance of the class).

Hidden variables

To achieve data hiding, it is often desirable to create "private" variables, which can be accessed only by the class itself. Jython provides a naming convention that makes accessing attributes and methods outside the class difficult. If you declare names of the form: *__xxx* or *__xxx_yyy* (that's two leading underscores), the Jython parser will automatically mangle (that is, add the class name to) the declared name, in effect creating hidden variables. For example:

```
class MyClass:
    __attr = 10              # private class attribute

    def method1(self):
        pass

    def method2(self, p1, p2):
        pass

    def __privateMethod(self, text):
        self.__text = text    # private attribute
```

Note that unlike C++ and the Java language, all references to instance variables must be qualified with `self`; there is no implied use of `this`.

The init method

The `__init__` method serves the role of an *instance constructor*. It is called whenever an instance is created. This method should be defined for all classes. Method `__init__` may take arguments. In Jython, and unlike in C++ or the Java language, all instance variables (also known as attributes or fields) are created dynamically by assignment. They should be defined (that is, assigned to) inside `__init__`. This ensures they are defined for subsequent methods to use. Some examples are as follows:

```
class Class1:
    def __init__ (self):             # no arguments
        self.data = []               # set implicit data

class Class2:
    def __init__ (self, v1, v2):     # 2 required arguments
        self.v1 = v1                 # set data from parameters
        self.v2 = v2

class Class3:
    def __init__ (self, values=None): # 1 optional argument
        if values is None: values = []
        self.values = values          # set data from parameter
```

The del method

If you allocate any resources in the `__init__` method (or any other method), you need to ensure they are released before the object is deallocated. The best way to do this is by using the `__del__` method. The `__del__` method is called just before the garbage collector deallocates the object. You should also provide a cleanup method (typically named `close`, `destroy`, or `dispose`) that can be called directly. Here's an example:

```
class Class:
    def __init__ (self, db):
        self.connection = db.getConnection()    # establish a connection
        self.connection.open()

    def __del__ (self):                         # cleanup at death
        self.close()

    def close(self):                            # cleanup
        if not self.connection is None and self.connection.isOpen():
            self.connection.close()             # release connection
        self.connection = None
```

Using classes as values

Classes can also be assigned to variables (including function arguments). This makes writing dynamic code based on classes quite easy, as you can see from the following generic class instance factory:

```
def instanceMaker(xclass, *args):
    return apply(xclass, args)
```

```
 :

x = instanceMaker(MyClass)   # same as: x = MyClass()
```

Inheritance

The ability to inherit from classes is a fundamental to object-oriented programming. Jython supports both single and multiple-inheritance. *Single inheritance* means there can be only one superclass; *multiple inheritance* means there can be more than one superclass.

Inheritance is implemented by subclassing other classes. These classes can be either other Jython classes or Java classes. Any number of pure-Jython classes or Java interfaces can be superclasses but only one Java class can be (directly or indirectly) inherited from. You are not required to supply a superclass.

Any attribute or method in a superclass is also in any subclass and may be used by the class itself or any client (assuming it is publicly visible). Any instance of a subclass can be used wherever an instance of the superclass can be used -- this is an example of *polymorphism*. These features enable reuse, rapid development, and ease of extension.

Below are some examples of inheritance:

```
class Class1: pass                # no inheritance

class Class2: pass

class Class3(Class1): pass        # single inheritance

class Class4(Class3,Class2): pass # multiple inheritance

from java import awt
from java import io

# inherit a Java class and interface and a Jython class
class MyPanel(awt.Panel, io.Serializable, Class2):
    :
```

The init method with inheritance

The `__init__` method of a subclass must call any `__init__` method defined for its superclass; this is not automatic. The two examples below demonstrate how the `__init__` method can be used with inheritance.

```
class Class1(SuperClass):
    def __init__ (self):                  # no arguments
        SuperClass.__init__(self)         # init my super-class
        self.data = []                    # set implicit data

class Class2(SuperClass):
    def __init__ (self, v1, v2):          # 2 required arguments
        SuperClass.__init__(self, v1)     # init my super-class with v1
        self.v2 = v2
```

And here are some examples of initializing with multiple inheritance:

```
class Class1(Super1, Super2):
    def __init__ (self):                  # no arguments
        Super1.__init__(self)             # init each super-class
        Super2.__init__(self)
        self.data = []                    # set implicit data

class Class2(Super1, Super2):
    def __init__ (self, v1, v2, v3):      # 3 required arguments
        # note you may do work before calling the super __init__ methods
        self.v3 = v3                      # set data from parameter
        Super1.__init__(self, v1)         # init each super-class
        Super2.__init__(self, v2)
```

Calling superclass methods

You can call any superclass method by qualifying it with the class name, as shown here:

```
class Class1:
    def method1 (self):
        :

class Class2(Class1):
    def method1 (self):           # override method1
        :
        Class1.method1(self)      # call my super-class method
        :

    def method2 (self):
        :

class Class3(Class2):
    def method1 (self):           # override method1
        :
        Class2.method1(self)      # call my super-class method
```

```
        :

    def method3 (self):
        :
```

Note that the secondary method definitions (in `Class2` and `Class3`) override the superclass definitions. There is no requirement that the subclass method call its superclass method; however, if it doesn't, then it must completely replace the function of the superclass method.

Calling methods

There are two syntaxes for calling methods (assuming you have an instance of `MyClass` referenced by variable `mci`):

- `mci.someMethod(...)`
- `MyClass.someMethod(mci, ...)`

The first form typically is used in class client coding while the second one is used more often in subclasses to call superclass methods.

Section 3. Advanced object-oriented programming

From theory to practice

In this section, we'll move from a conceptual overview of object-oriented programming in Jython to a more advanced discussion, incorporating topics such as operator overloading, special attributes, and introspection.

Special attributes

Jython classes provide support for several special attributes. The most significant are shown below:

Name	**Role**	**Comment(s)**
`__dict__`	The object's writeable attributes	Can be used to introspect the attributes of an object
`__class__`	The class of an object	Access the class of the object (similar to `x.getClass()` in Java coding)
`__bases__`	A tuple of the immediate superclasses of the object	Can be used to introspect the superclasses of the object

---

Changing the class of an existing instance

In Jython, unlike most other languages, you can change the class of an existing instance. Doing this changes the methods you can then use on the instance to the methods of the new class but not any of its pre-existing fields. For example, to change the class of an instance, assign the new class to the `__class__` special attribute (see Special attributes on page 13), as shown below:

```
x = SomeClass()
print isinstance(x, SomeClass)       # prints: 1 (true)
print isinstance(x, SomeOtherClass)  # prints: 0 (false)
 :
# change the class (that is, the type) of the instance here
x.__class__ = SomeOtherClass
print isinstance(x, SomeClass)       # prints: 0 (false)
```

```
print isinstance(x, SomeOtherClass)  # prints: 1 (true)

y = SomeOtherClass()
print x.__class__ == y.__class__     # prints: 1 (true)
```

After this change, the *x* instance will support the methods of `SomeOtherClass`, not `SomeClass` as it did previously. Take care when changing the class of an object that the instance has the right attributes for the new class.

Introspecting attributes example

Here's a practical example using special attributes (see Special attributes on page 13). The module `printclass.py` can introspect classes and instances to display their attributes and methods. I'll talk about introspection a little later, or you can check Introspection on page 16 . You can also see String operations and functions on page 46 and Appendix K: Built-in functions on page 95 to learn more about the functions used below. For right now, just focus on the use of the `callable` function, the `vars` function (which implicitly uses the `__dict__` attribute) and the `__bases__` attribute.

```
__any__ = ['getMembers', 'printObject']

def addMember (list, item):
    if not item in list:
        list.append(item)

def getMembers (obj, memtype="attrs"):
    """ Get all the members (of memtype) of the object. """
    members = []
    for name, value in vars(obj).items():
        try:
            item = obj.__name__, name, value
        except:
            item = "<instance>", name, value
        if  memtype.lower().startswith("attr"):
            if not callable(value):
                addMember(members, item)
        elif memtype.lower().startswith("meth"):
            if callable(value):
               addMember(members, item)
        elif memtype.lower() == "all":
            addMember(members, item)
    try:
        for base in obj.__bases__:
            members.extend(getMembers(base, memtype))
    except:
        pass
    return members

import sys
```

```
def printObject (obj, stream=sys.stdout):
    """ Print all the members of the object. """
    members = getMembers(obj, "attrs")
    members.sort()
    print >>stream, "Attributes:"
    for objname, memname, value in members:
        print >>stream, "  %s.%s" % (objname, memname)

    members = getMembers(obj, "methods")
    members.sort()
    print >>stream, "Methods:"
    for objname, memname, value in members:
        print >>stream, "  %s.%s" % (objname, memname)
```

Introspecting attributes example testcase

The following code uses the functions in the previous panel to introspect the `UserList` class. See Operator overloading on page 21 for the definition of the `UserList` class.

```
if __name__ == "__main__":

    from UserList import UserList

    class MyClass(UserList):
        def __init__ (self, x, y):
            UserList.__init__(self)
            self.__x = x
            self.__y = y

        def method1 (self):
            return self.x + self.y

        def method2 (self, x, y):
            return  self.x + self.y + x + y

    print "For class:", `MyClass`
    printObject(MyClass)
    print

    aMyClass = MyClass(1, 2)
    aMyClass.extend([1,2,3,4])
    print "For instance:", `aMyClass`
    printObject(aMyClass)
```

Output of get members

The following output (reformatted into multiple columns to save space) is the result of running the main code from the above module. Notice that the private fields and methods (see Hidden variables on page 8) have mangled names.

```
For class: <class __main__.MyClass at 28921555>
Attributes:             Methods:
  MyClass.__doc__         MyClass.__init__          UserList.__len__
  MyClass.__module__      MyClass.method1           UserList.__lt__
  UserList.__doc__        MyClass.method2           UserList.__mul__
  UserList.__module__     UserList._UserList__cast  UserList.__ne__
                          UserList.__add__          UserList.__radd__
                          UserList.__cmp__          UserList.__repr__
                          UserList.__contains__     UserList.__rmul__
                          UserList.__delitem__      UserList.__setitem__
                          UserList.__delslice__     UserList.__setslice__
                          UserList.__eq__           UserList.append
                          UserList.__ge__           UserList.count
                          UserList.__getitem__      UserList.extend
                          UserList.__getslice__     UserList.index
                          UserList.__gt__           UserList.insert
                          UserList.__iadd__         UserList.pop
                          UserList.__imul__         UserList.remove
                          UserList.__init__         UserList.reverse
                          UserList.__le__           UserList.sort

For instance: [1, 2, 3, 4]
Attributes:
  <instance>._MyClass__x
  <instance>._MyClass__y
  <instance>.data

Methods:
```

Note that methods and class attributes reside with classes and instance attributes reside with instances. Yet all the class's methods can be applied to each instance.

Introspection

You will often need to determine, at runtime, the characteristics of an object. We call this *introspecting* the object. The Java platform offers introspection services via the `java.lang.Class` class and classes in the `java.lang.reflect` package. While powerful, these APIs are somewhat difficult to use. As you probably already suspected, Jython offers a simpler approach to introspection.

In Jython, we can use the `dir` and `vars` functions to examine the bindings for any object, such as modules, functions, classes, sequences, maps, and more. To better understand how this works, consider the following example. The output has been inserted (and reformatted) after the `print` statements prefixed with "..." for easier reading. The `dir` function returns only the binding names, while the `vars` function returns the names and values; thus, when the same names are returned by both functions, we need use only the `vars` function, as shown below:

```
#-- empty start --
print "vars:", vars()
...vars: {'__doc__': None, '__name__': '__main__'}

x = 1
y = 2
z = 3
l = [x, y, z]
d = {x:"xxxx", y:"yyyy", z:"zzzz"}

#-- locals variables --
print x, y, z, l, d
...1 2 3 [1, 2, 3] {3: 'zzzz', 2: 'yyyy', 1: 'xxxx'}

#-- plus locals variables --
print "vars:", vars()
...vars: {'__name__': '__main__', 'x': 1, \
... 'd': {3: 'zzzz', 2: 'yyyy', 1: 'xxxx'}, '__doc__': None, \
... 'y': 2, 'l': [1, 2, 3], 'z': 3}

import sys

#-- plus import --
print "vars:", vars()
...vars: {'__name__': '__main__', 'z': 3, 'l': [1, 2, 3], \
... '__doc__': None, 'y': 2, 'x': 1, 'sys': sys module, \
... 'd': {3: 'zzzz', 2: 'yyyy', 1: 'xxxx'}}

#-- sys import --
print "vars sys:", vars(sys)
...vars sys: {'classLoader': \
...     <beanProperty classLoader type: java.lang.ClassLoader at 31845755>,
...     ... many values removed ...,
... 'warnoptions': <reflected field public static \
...     org.python.core.PyList \
...     org.python.core.PySystemState.warnoptions at 1024901>}

del x, y, z

#-- post delete --
print "vars:", vars()
...vars: {'__name__': '__main__', 'l': [1, 2, 3], '__doc__': None, \
... 'sys': sys module, 'd': {3: 'zzzz', 2: 'yyyy', 1: 'xxxx'}}

def func (x, y):
    return x, y

class MyClass ():
    def __init__ (self, x, y):
        self.__x = x
        self.__y = y

    def method1 (self):
        return self.x + self.y
```

```
    def method2 (self, x, y):
        return  self.x + self.y + x + y

#-- plus function and class --
print "vars:", vars()
....vars: {'func': <function func at 21569784>, '__name__': '__main__', \
...  'l': [1, 2, 3], '__doc__': None, \
.... 'MyClass': <class __main__.MyClass at 1279942>, \
...  'sys': sys module, 'd': {3: 'zzzz', 2: 'yyyy', 1: 'xxxx'}}

#-- function --
print "dir: ", dir(func)     # **** dir and vars different here ****
print "vars:", vars(func)
...dir:  ['__dict__', '__doc__', '__name__', 'func_closure', \
... 'func_code', 'func_defaults', 'func_doc', 'func_globals', 'func_name']
...vars: None

#-- class --
print "vars:", vars(MyClass)
...vars: {'__doc__': None, '__init__': <function __init__ at 17404503>, \
... 'method2': <function method2 at 23511968>, '__module__': '__main__', \
... 'method1': <function method1 at 28670096>}

myclass = MyClass(1, 2)

#-- instance --
print "myclass:", myclass
print "vars:", vars(myclass)
...myclass: <__main__.MyClass instance at 19014134>
...vars: {'_MyClass__y': 2, '_MyClass__x': 1}
```

Note that `dir(x)` is generally equivalent to `x.__dict__.keys()` and `vars(x)` is generally equivalent to `x.__dict__`.

Additional functions for introspection

The attributes described in Special attributes on page 13 allow additional introspection of classes. In particular you can use the `__dict__` attribute to determine the methods in a class and the fields in an instance.

In addition to `dir` and `vars`, Jython provides several more functions for introspecting classes and instances, as follows:

Function	Comment(s)
`hasattr(obj, name)`	Tests to see if the named attribute exists
`getattr(obj, name {, default})`	Gets the named attribute if it exists; else default is returned (or an exception is raised if no default is provided)
`setattr(obj, name,`	Sets the named attribute's value

value)	
delattr(obj, name)	Removes the named attribute

See Appendix K: Built-in functions on page 95 to learn more about these functions.

Abstract classes

Abstract classes are classes in which some or all of the methods are missing or have incomplete definitions. A subclass must be created to provide or complete these method definitions. *Concrete* classes are not abstract (that is, all the methods are complete). So far we have been working only with concrete classes. Abstract classes are created to facilitate reuse. They provide a partial implementation of a design that you can complete or extend by subclassing them.

To get a better understanding of how this works, we will create a simple abstract command framework that supports command `do`, `undo`, and `redo` actions. Commands are defined in (sub)classes and can be added easily by creating new `do_...` and `undo_...` methods. We access these methods via introspection, as discussed in the previous panels.

An abstract command framework

Here's the example abstract command framework:

```
class CommandProcessor:   # an abstract class
    """ Process Commands. """

    def __init__ (self):
        self.__history = []
        self.__redo = []

    def execute (self, cmdName, *args):
        """ Do some command """
        self.__history.append( (cmdName, args) )
        processor = getattr(self, "do_%s" % cmdName, None)
        if processor:
            return processor(*args)
        else:
            raise NameError, "cannot find do_%s" % cmdName

    def undo (self, count=1):
        """ Undo some (or all) commands in LIFO order """
        self.__redo = []
```

```
        while count > 0 and len(self.__history) > 0:
            cmdName, args = self.__history.pop()
            count -= 1
            processor = getattr(self, "undo_%s" % cmdName, None)
            if processor:
                self.__redo.append( (cmdName, args) )
                processor(*args)
            else:
                raise NameError, "cannot find undo_%s" % cmdName

    def redo (self, count=1):
        """ Redo some (or all) undone commands """
        while count > 0 and len(self.__redo) > 0:
            cmdName, args = self.__redo.pop()
            count -= 1
            processor = getattr(self, "do_%s" % cmdName, None)
            if processor:
                processor(*args)
            else:
                raise NameError, "cannot find do_%s" % cmdName
```

Note: This example is based on code from *Jython Essentials* by Samuele Pedroni and Noel Rappin (see Resources on page 73 for more information).

A test case for the example framework

Here's a test case for the example abstract command framework:

```
class MyProcessor (CommandProcessor):  # a concrete subclass
    def __init__ (self):
        CommandProcessor.__init__(self)

    def do_Cmd1 (self, args):
        print "Do Command 1:", args

    def do_Cmd2 (self, args):
        print "Do Command 2:", args

    def do_Cmd3 (self, args):
        print "Do Command 3:", args

    def undo_Cmd1 (self, args):
        print "Undo Command 1:", args

    def undo_Cmd2 (self, args):
        print "Undo Command 2:", args

    def undo_Cmd3 (self, args):
        print "Undo Command 3:", args

mp = MyProcessor()
```

```
print "execute:" ; mp.execute("Cmd1", None)
print "execute:" ; mp.execute("Cmd2", (1,2,3))
print "execute:" ; mp.execute("Cmd3", "Hello")
print "undo:   " ; mp.undo(2)
print "redo:   " ; mp.redo(2)

print "execute:", ;mp.execute("BadCmd", "Hello")
```

The framework with the given test case produces the following output:

```
execute:
Do Command 1: None
execute:
Do Command 2: (1, 2, 3)
execute:
Do Command 3: Hello
undo:
Undo Command 3: Hello
Undo Command 2: (1, 2, 3)
redo:
Do Command 2: (1, 2, 3)
Do Command 3: Hello
execute:
Traceback (innermost last):
  File "cmdproc.py", line 63, in ?
  File "cmdproc.py", line 15, in execute
NameError: cannot find do_BadCmd
```

Operator overloading

Like C++, but unlike the Java language, Jython allows many of the standard language operators to be overloaded by classes. This means classes can define a specific meaning for the language operators. Jython also allows classes to emulate built-in types like numbers, sequences, and maps. To learn more about emulation see Appendix B: Common overloaded operators and methods on page 76 .

In the example that follows, we'll use the standard Jython `UserList` class definition to show an example of operator overloading in practice. `UserList` is a class that wraps a list and behaves as a list does. Most of its function is *delegated* (passed on to) its contained list, called `data`. In a more realistic example, these overloaded functions would be implemented to access some other store, such as a disk file or a database.

```
class UserList:
    def __init__(self, initlist=None):
        self.data = []
        if initlist is not None:
            if   type(initlist) == type(self.data):
                self.data[:] = initlist
            elif isinstance(initlist, UserList):
```

```
                self.data[:] = initlist.data[:]
            else:
                self.data = list(initlist)

    def __cast(self, other):
        if isinstance(other, UserList): return other.data
        else:                           return other

    #  `self`, repr(self)
    def __repr__(self): return repr(self.data)

    #  self < other
    def __lt__(self, other): return self.data <  self.__cast(other)

    #  self <= other
    def __le__(self, other): return self.data <= self.__cast(other)

    #  self == other
    def __eq__(self, other): return self.data == self.__cast(other)

    #  self != other, self <> other
    def __ne__(self, other): return self.data != self.__cast(other)

    #  self > other
    def __gt__(self, other): return self.data >  self.__cast(other)

    #  self >= other
    def __ge__(self, other): return self.data >= self.__cast(other)

    #  cmp(self, other)
    def __cmp__(self, other):
        raise RuntimeError, "UserList.__cmp__() is obsolete"

    #  item in self
    def __contains__(self, item): return item in self.data

    #  len(self)
    def __len__(self): return len(self.data)

    #  self[i]
    def __getitem__(self, i): return self.data[i]

    #  self[i] = item
    def __setitem__(self, i, item): self.data[i] = item

    #  del self[i]
    def __delitem__(self, i): del self.data[i]

    #  self[i:j]
    def __getslice__(self, i, j):
        i = max(i, 0); j = max(j, 0)
        return self.__class__(self.data[i:j])

    #  self[i:j] = other
    def __setslice__(self, i, j, other):
        i = max(i, 0); j = max(j, 0)
        if   isinstance(other, UserList):
            self.data[i:j] = other.data
```

```
        elif isinstance(other, type(self.data)):
            self.data[i:j] = other
        else:
            self.data[i:j] = list(other)

    #  del self[i:j]
    def __delslice__(self, i, j):
        i = max(i, 0); j = max(j, 0)
        del self.data[i:j]

    #  self + other   (join)
    def __add__(self, other):
        if   isinstance(other, UserList):
            return self.__class__(self.data + other.data)
        elif isinstance(other, type(self.data)):
            return self.__class__(self.data + other)
        else:
            return self.__class__(self.data + list(other))

    #  other + self   (join)
    def __radd__(self, other):
        if   isinstance(other, UserList):
            return self.__class__(other.data + self.data)
        elif isinstance(other, type(self.data)):
            return self.__class__(other + self.data)
        else:
            return self.__class__(list(other) + self.data)

    #  self += other  (join)
    def __iadd__(self, other):
        if   isinstance(other, UserList):
            self.data += other.data
        elif isinstance(other, type(self.data)):
            self.data += other
        else:
            self.data += list(other)
        return self

    #  self * other   (repeat)
    def __mul__(self, n):
        return self.__class__(self.data*n)
    __rmul__ = __mul__

    #  self *= other  (repeat)
    def __imul__(self, n):
        self.data *= n
        return self

    # implement "List" functions below:

    def append(self, item): self.data.append(item)

    def insert(self, i, item): self.data.insert(i, item)

    def pop(self, i=-1): return self.data.pop(i)

    def remove(self, item): self.data.remove(item)
```

```
    def count(self, item): return self.data.count(item)

    def index(self, item): return self.data.index(item)

    def reverse(self): self.data.reverse()

    def sort(self, *args): apply(self.data.sort, args)

    def extend(self, other):
        if isinstance(other, UserList):
            self.data.extend(other.data)
        else:
            self.data.extend(other)
```

Nested classes

Like functions, classes can be nested. Nested classes in Jython work similarly to static inner classes in the Java language. Here's an example:

```
class MyDataWrapper:
    class Data: pass    # inner data structure class

    def __init__ (self):
        self.data = Data()

    def set (self, name, value):
        setattr(self.data, name, value)

    def get (self, name, default=None):
        return getattr(self.data, name, default)
```

Section 4. Debugging Jython

Using print statements for debugging

Like any programming language, Jython supports the use of `print` statements for debugging. To implement this debugging solution, we simply add a `print` statement to a program, run the program, and examine the generated output for clues to the bugs. While very basic, this debugging solution is in many cases completely satisfactory.

Here's an example `print` statement for debugging.

```
:
def myFunc(x):
    print "x at entry:", x
      :
    print "x at exit:", x
    return x
:

z = myFunc(20)
```

The Jython debugger

For the times when the `print`-statement solution isn't sufficient for your debugging needs, Jython provides a simple, command-line debugger similar to the `jdb` debugger for the Java platform. The Jython debugger is written entirely in Jython and can thus be easily examined or extended. In addition, Jython provides a set of abstract base debugging classes to allow other debuggers, such as a GUI debugger, to be built on this framework.

To launch the debugger run the following command:

```
c:\>jython c:\jython-2.1\lib\pdb.py <test_module>.py
```

An example Jython debugging session

Debugger commands are enterred after the debugger prompt "(Pdb)." Here's an example debugging session using the `factor.py` module (see The factorial engine: factor.py on page 67):

```
C:\Articles>jython \jython-2.1\lib\pdb.py factor.py
> C:\Articles\<string>(0)?()
(Pdb) step
> C:\Articles\<string>(1)?()
(Pdb) step
> C:\Articles\factor.py(0)?()
(Pdb) list 67
 62             try:
 63                 print "For", value, "result =",
fac.calculate(value)
 64             except ValueError, e:
 65                 print "Exception -", e
 66
 67         doFac(-1)
 68         doFac(0)
 69         doFac(1)
 70         doFac(10)
 71         doFac(100)
 72         doFac(1000)
(Pdb) tbreak 67
Breakpoint 1 at C:\Articles\factor.py:67
(Pdb) continue
factor.py running...
Deleted breakpoint 1
> C:\Articles\factor.py(67)?()
-> doFac(-1)
(Pdb) next
For -1 result = Exception - only positive integers supported: -1
> C:\Articles\factor.py(68)?()
-> doFac(0)
(Pdb) next
For 0 result = 1
> C:\Articles\factor.py(69)?()
-> doFac(1)
(Pdb) next
For 1 result = 1
> C:\Articles\factor.py(70)?()
-> doFac(10)
(Pdb) next
For 10 result = 3628800
> C:\Articles\factor.py(71)?()
-> doFac(100)
(Pdb) next
For 100 result =
93326215443944152681699238856266700490715968264381621468592963895217599
99322991560894146397615651828625
3697920827223758251185210916864000000000000000000000000
> C:\Articles\factor.py(72)?()
-> doFac(1000)
(Pdb) next
For 1000 result = 402387260077 ... many other digits deleted ...
0000000000000000000000
--Return--
> C:\Articles\factor.py(72)?()->None
-> doFac(1000)
(Pdb) next
--Return--
```

```
> C:\Articles\<string>(1)?()->None
(Pdb) next
C:\Articles>
```

To learn more about debugging with the Jython debugger, see Appendix C: Jython debugger commands on page 79 .

Jython profiler

Sometimes you may notice that a Jython program runs longer than you expect. You can use the Jython profiler to find out what sections of the program take the longest time and optimize them. The profiler will let you profile entire programs or just individual functions.

Here's an example run, profiling the `factor.py` program (see The factorial engine: factor.py on page 67):

```
c:\>jython \jython-2.1\lib\profile.py \articles\factor.py

\articles\factor.py running...
For -1 result = Exception - only positive integers supported: -1
For 0 result = 1
For 1 result = 1
For 10 result = 3628800
For 100 result =
93326215443944152681699238856266700490715968264381621468592963895217599
99322991560894146397615651828625369792082722375825118521091686400000000
0000000000000000
For 1000 result = 402387260077 ... many other digits deleted ...
0000000000000000000000

         237 function calls (232 primitive calls) in 0.250 CPU seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.130    0.130    0.240    0.240 <string>:0(?)
        1    0.000    0.000    0.110    0.110 factor.py:0(?)
      220    0.010    0.000    0.010    0.000 \
factor.py:27(fireListeners)
        6    0.060    0.010    0.070    0.012 factor.py:34(calculate)
        1    0.000    0.000    0.000    0.000 factor.py:5(Factorial)
        1    0.000    0.000    0.000    0.000 factor.py:6(__init__)
      6/1    0.040    0.007    0.110    0.110 factor.py:61(doFac)
        1    0.010    0.010    0.250    0.250 \
profile:0(execfile('\\articles\\factor.py'))
        0    0.000             0.000          profile:0(profiler)
```

From this run you can see that (besides the initial startup code) most of the program time is being used by the `calculate` function. For more information on profiling

Jython see the *Python Reference Manual*, available in Resources on page 73 .

Assertions

Like C and the Java language (as of version 1.4), Jython supports assertions. *Assertions* are conditions that must be true for the program to work correctly; if they are not true the program may behave unpredictably. Often they are used to validate input values to functions. Jython's support for assertions comes in the form of the following `assert` statement:

```
assert expression {, message}
```

Note that `expression` is any Jython expression; if it is false an `exceptions.AssertionError` exception is raised. If `message` is provided, it becomes the message associated with the exception. For example:

```
:
def myFunc(x):
    assert x >= 0, "argument %r must be >= 0" % x
    return fac(x)
:
z = myFunc(20)          # no exception raised

z = myFunc(-1)          # AssertionError raised
```

Section 5. Java support in Jython

Using Java services in Jython code

One of Jython's most powerful features is its ability to interface with Java code. A Jython program can create instances of any Java class and call any method on any Java instance. Jython can also subclass Java classes, allowing Java code to call Jython code. Jython makes calling Java methods very easy by making strong but transparent use of the Java Reflection API (package `java.lang.reflect`).

To complete this section of the tutorial, you need to be familiar with the Java language and select Java runtime APIs. You should understand the basic notions of object-oriented programming on the Java platform, as well as being familiar with the Java data types, classes, threads, and the services in the `java.lang`, `java.util`, `java.io` and `javax.swing` packages.

Note: Because the reflection APIs have been highly optimized in version 1.4, Jython runs much faster on Java version 1.4 and above.

Calling Jython from Java code

As shown in Inheritance on page 10 , a Jython class can subclass Java classes. Subclassing makes it very easy to extend Java classes (such as GUI components). This allows Java code to call Jython code without realizing it is Jython code. It also makes it possible to implement in Jython classes used by other Java code, as shown in the following example:

```
from java import util
class MyArray(util.ArrayList):  # subclass a Java class
    :
    def get (self, index):      # override the get method
        "@sig public java.lang.Object get(int index)"
      if 0 <= index < self.size:
            return util.ArrayList.get(self, index)
        return None             # OutOfBounds now returns null
```

After being compiled by `jythonc` the above class can be used in Java code anywhere an `java.util.ArrayList` instance can be used. Note that when calling a superclass method, the *self* value is passed as an argument.

Calling Java classes from Jython

In addition to subclassing Java classes it is also possible to access Java classes directly in Jython. For example, this code sequence:

```
from java.util import Date

 :

d = Date()   # now
print d, d.time, d.getTime()
```

will produce the following output:

```
Tue Dec 02 14:44:02 CST 2003 1070397842496 1070397842496
```

Using JavaBean properties from Jython

In the example from Calling Java classes from Jython on page 30 you may have noticed that the expressions `d.time` and `d.getTime()` produce the same result. This is because they do the same thing. Jython has a very convenient feature that makes JavaBean properties appear as Jython attributes. JavaBean properties are defined by (typically) matching pairs of Java methods of the following form, where `<type>` is the type of the property and `<name>` is the name of the property.:

```
<type> get<name>()

-- and --

void set<name>(<type> value)
```

For example the Java methods `long getTime() { ... }` and `void setTime(long t) { ... }` define the `long` property *time*. Thus a Jython reference `d.time` is automatically and dynamically converted into the Java expression `d.getTime()`.

Jython can also set properties, thus `d.time = 1000000L` is allowed. The Jython reference `d.time =` *`value`* is automatically and dynamically converted into the Java expression `d.setTime(`*`value`*`)`. Once this change is applied, the print statement from Calling Java classes from Jython on page 30 results in the following:

```
Wed Dec 31 18:01:40 CST 1969 100000 100000
```

Calling methods on Java objects

It is very easy to call methods on Java objects; just call them like they are Jython methods. Jython automatically maps parameter and return values to and from Jython and Java types. For example, here is a short sequence of Jython that uses Java classes and methods extensively:

```
1: from javax import swing
2: import sys
3:
4: f = swing.JFrame(sys.argv[1], size=(200,200),
5:                  defaultCloseOperation=swing.JFrame.EXIT_ON_CLOSE)
6: f.contentPane.add(swing.JLabel(sys.argv[2]))
7: f.visible = 1
```

This code sequence creates and shows a GUI frame window. The script's first command-line argument becomes the title and the second the content text. Line 4 creates the frame, passing in the title, the desired size, and a close action. The `size` and `defaultCloseOperation` parameters are properties as described above and, as such, may be (quite conveniently) set in the `JFrame`'s constructor when invoked from a Jython program. The title is set as a parameter of the `JFrame`'s equivalent of the `__init__` method. Line 6 accesses the `JFrame`'s `contentPane` property and calls its `add` method to add a `JLabel` to show the second argument. Line 7 makes the frame visible by setting its `visible` property to `1` (true).

A sample of this GUI is shown below:

Overriding Java methods and properties

As shown in Calling Jython from Java code on page 29 , when overriding Java methods in classes that can be called from the Java language, you need to provide signature information. This is done via documentation comments. The first line of the comment, if it starts with `"@sig"`, is used as a directive to the `jythonc` program (discussed in Part 1) to generate a Java-compatible method signature. For example, the comment below describes the `get` method using the Java language's declaration syntax. In signatures types must be fully qualified.

```
"@sig public java.lang.Object get(int index)"
```

Jython does not support overloaded methods, which are methods with the same name but with differing number and/or types of arguments. Instead, Jython supports defaulted arguments and variable number of arguments, which can create a problem if you inherit from a Java class that uses overloading and you want to override the overloaded methods. In Jython, you must define the base method and accept a varying number of arguments. Consider the (rather impractical) example of an `InputStream` that always returns a blank:

```
from java import io

class AlwaysBlank(io.InputStream):
    # covers all forms of read(...)
    def read(self, *args):
        if len(args) > 0:
            # covers forms: int read(byte[])
            #               int read(byte[], int off, int len)
            return apply(io.InputStream.read, (self,) + args)
        else:
            # covers form: int read()
            return ord(' ')
```

This code is based on an example from the Jython home page.

Java arrays from Jython

Jython supports the creation of Java-style array objects. *Arrays* are used primarily to pass arrays to and return arrays from Java methods, but they are general purpose and can be used in pure Jython code. Array elements are typed using Java base and class types. Arrays act much like Jython lists but they cannot change length.

Array support is provided by the `jarray` module. The two functions in the `jarray` module, `zeros` and `array`, are used to create arrays. The array function maps a Jython sequence to a Java array. Some examples are as follows:

```
from jarray import zeros, array
from java import util
from javax import swing

a1 = zeros(100, 'i')            # an array of 100 int 0s
a2 = array([1,2,10,-5,7], 'i')  # an array of ints as listed

# an array of doubles 0.0 to 49.0
a3 = array([i * 1.0 for i in range(50)], 'd')

a4 = zeros(10, util.Map)        # an array of 10 null Maps
a5 = array((swing.JFrame("F1"), # an array of 3 JFrames
            swing.JFrame("F2"),
            swing.JFrame("F3")), swing.JFrame)
a6 = array("Hello", 'c')        # an array of characters
```

See Appendix A: Character codes for array types on page 76 for a listing of character codes for array types.

Section 6. Java thread support in Jython

Java threads

The Java runtime makes extensive use of threads, which it uses to handle GUI events, to perform asynchronous I/O, to implement asynchronous processing, and so on.

It's easy to create Java threads in Jython: just create instances of `java.lang.Thread` and subclasses of `java.lang.Runnable`. For an example, see The GUI: fgui.py on page 69 . You can also create threads out of Jython functions by using the `thread` module and functions of the following form:

```
start_new_thread(function, args)

-- and --

exit()
```

The `start_new_thread` function runs the `function` argument in a new Java thread, passing the *args* tuple value to the function. The `exit` function can be used in the thread to end it (generally as the target of an `if` statement).

Java synchronization

When developing multithreaded programs using Java or Jython threads, it is sometimes necessary to create synchronized functions (or methods). *Synchronized functions* are functions that can only be called from one thread at a time; meaning that other threads are prevented from entering the function until the first thread exits. Jython provides the `synchronized` module and two functions to create synchronized functions. The functions are of the following form:

```
make_synchronized(function)

-- and --

apply_synchronized(syncobj, function, pargs {, kwargs})
```

The `make_synchronized` function permanently synchronizes the `function` argument. The `apply_synchronized` function temporarily synchronizes on `syncobj` and then calls the `function` argument.

Example: Using make_synchronized

Using `make_synchronized` to signal events is quite straightforward, as shown below:

```
from synchronize import *
from java import lang

# define synchronization helpers

def waitForSignal (monitor):
    """ Wait until the monitor is signaled. """
    lang.Object.wait(monitor)
# replace with synchronized version; syncs on 1st argument
waitForSignal = make_synchronized(waitForSignal)

def notifySignal (monitor):
    """ Signal monitor. """
    lang.Object.notifyAll(monitor)
# replace with synchronized version; syncs on 1st argument
notifySignal = make_synchronized(notifySignal)

class Gui:           # GUI support
    :
    def doExit (self):
        self.visible = 0
        notifySignal(self)

if __name__ == "__main__":     # main code
    :
    gui = Gui()
    :
    print "Waiting until GUI exit requested..."
    waitForSignal(gui)
    print "Done"
```

A Jython threading example

Here's an example of the use of Jython threads. The example shows a set of producer and consumer threads sharing access to a common buffer. We'll start with the definition of the shared buffer, as shown below.

```
""" A Jython Thread Example. """

from java import lang
from synchronize import *
from thread import start_new_thread
from sys import stdout
```

```
def __waitForSignal (monitor):
    apply_synchronized(monitor, lang.Object.wait, (monitor,))

def __signal (monitor):
    apply_synchronized(monitor, lang.Object.notifyAll, (monitor,))

def __xprint (stream, msg):
    print >>stream, msg

def xprint (msg, stream=stdout):
    """ Synchronized print. """
    apply_synchronized(stream, __xprint, (stream, msg))

class Buffer:
    """ A thread-safe buffer. """

    def __init__ (self, limit=-1):
        self.__limit = limit    # the max size of the buffer
        self.__data = []
        self.__added = ()       # used to signal data added
        self.__removed = ()     # used to signal data removed

    def __str__ (self):
        return "Buffer(%s,%i)" % (self.__data, self.__limit)

    def __len__ (self):
        return len(self.__data)

    def add (self, item):
        """ Add an item. Wait if full. """
        if self.__limit >= 0:
            while len(self.__data) > self.__limit:
                __waitForSignal(self.__removed)
        self.__data.append(item);
        xprint("Added: %s" % item)
        __signal(self.__added)

    def __get (self):
        item = self.__data.pop(0)
        __signal(self.__removed)
        return item

    def get (self, wait=1):
        """ Remove an item. Wait if empty. """
        item = None
        if wait:
            while len(self.__data) == 0:
                __waitForSignal(self.__added)
            item = self.__get()
        else:
            if len(self.__data) > 0: item = self.__get()
        xprint("Removed: %s" % item)
        return item
    get = make_synchronized(get)
```

Producer and consumer definitions

The next step in the example is to take a look at the producer and consumer definitions, shown here:

```
class Producer:
    def __init__ (self, name, buffer):
        self.__name = name
        self.__buffer = buffer

    def __add (self, item):
        self.__buffer.add(item)

    def __produce (self, *args):
        for item in args:
            self.__add(item)

    def produce (self, items):
        start_new_thread(self.__produce, tuple(items))

class Consumer:
    def __init__ (self, name, buffer):
        self.__name = name
        self.__buffer = buffer

    def __remove (self):
        item = self.__buffer.get()
        return item

    def __consume (self, count):
        for i in range(count):
            self.__remove()

    def consume (self, count=1):
        start_new_thread(self.__consume, (count,))
```

An trial run of the threading example

And finally, here's a trial run of the example code:

```
# all producers and consumer share this one
buf = Buffer(5)

p1 = Producer("P1", buf)
p2 = Producer("P2", buf)
p3 = Producer("P3", buf)
p4 = Producer("P4", buf)
c1 = Consumer("C1", buf)
```

```
c2 = Consumer("C2", buf)

# create 6 items
p1.produce(["P1 Message " + str(i) for i in range(3)])
p2.produce(["P2 Message " + str(i) for i in range(3)])

# consume 20 items
for i in range(5):
    c1.consume(2)
    c2.consume(2)

# create 20 more items
p3.produce(["P3 Message " + str(i) for i in range(10)])
p4.produce(["P4 Message " + str(i) for i in range(10)])

# consume 4 items
c1.consume(2)
c2.consume(2)

# let other threads run
lang.Thread.currentThread().sleep(5000)

xprint("Buffer has %i item(s)left" % len(buf))
```

Output of the example

The producer consumer example produces the following results (wrapped to two columns to save space):

```
Added: P1 Message 0       Added: P3 Message 7
Added: P1 Message 1       Removed: P3 Message 7
Added: P1 Message 2       Added: P3 Message 8
Added: P2 Message 0       Removed: P3 Message 8
Added: P2 Message 1       Added: P3 Message 9
Added: P2 Message 2       Removed: P3 Message 9
Removed: P1 Message 0     Added: P4 Message 0
Removed: P1 Message 1     Removed: P4 Message 0
Removed: P1 Message 2     Added: P4 Message 1
Removed: P2 Message 0     Removed: P4 Message 1
Removed: P2 Message 1     Added: P4 Message 2
Removed: P2 Message 2     Removed: P4 Message 2
Added: P3 Message 0       Added: P4 Message 3
Removed: P3 Message 0     Removed: P4 Message 3
Added: P3 Message 1       Added: P4 Message 4
Removed: P3 Message 1     Added: P4 Message 5
Added: P3 Message 2       Added: P4 Message 6
Removed: P3 Message 2     Added: P4 Message 7
Added: P3 Message 3       Added: P4 Message 8
Removed: P3 Message 3     Added: P4 Message 9
Added: P3 Message 4       Removed: P4 Message 4
Removed: P3 Message 4     Removed: P4 Message 5
Added: P3 Message 5       Removed: P4 Message 6
```

```
Removed: P3 Message 5     Removed: P4 Message 7
Added: P3 Message 6       Buffer has 2 item(s)left
Removed: P3 Message 6
```

Section 7. Interfacing with Java services

Creating the interface

Often you will need to use Java services from within Jython code. In these cases, you can either do it openly each time you need to use a given service, or you can wrap the Java services in a Jython library function and use that function in your Jython code.

The second option is recommended because it encapsulates and abstracts the Java code.

Wrapping Java services in Jython

As an example of how you might wrap a Java service in a Jython library function, we'll take a look at the `JavaUtils.py` module excerpts. The `JavaUtils` module is introduced by the code below. See *Part 1* of this tutorial to refresh your memory about modules.

```
""" This module defines several functions to ease interfacing with Java code."""

from types import *

from java import lang
from java import util
from java import io

# only expose these
__all__ = ['loadProperties', 'getProperty',
           'mapToJava', 'mapFromJava', 'parseArgs']
```

Accessing Java properties files

You will often need to access Java properties files to get configuration information. Jython lets you use the `loadProperties` and `getProperty` functions for this, as shown below:

```
def loadProperties (source):
    """ Load a Java properties file into a Dictionary. """
    result = {}
    if type(source) == type(''):    # name provided, use file
```

```
        source = io.FileInputStream(source)
    bis = io.BufferedInputStream(source)
    props = util.Properties()
    props.load(bis)
    bis.close()
    for key in props.keySet().iterator():
        result[key] = props.get(key)
    return result

def getProperty (properties, name, default=None):
    """ Gets a property. """
    return properties.get(name, default)
```

Properties file example

So, for example, if you were to use the functions from Accessing Java properties files on page 40 as shown below

```
import sys
file = sys.argv[1]
props = loadProperties(file)
print "Properties file: %s, contents:" % file
print props
print "Property %s = %i" % ('debug', int(getProperty(props, 'debug', '0')))
```

with the properties file content of

```
# This is a test properties file
debug = 1
error.level = ERROR
now.is.the.time = false
```

then the resulting output would be:

```
Properties file: test.properties, contents:
{'error.level': 'ERROR', 'debug': '1', 'now.is.the.time': 'false'}
Property debug = 1
```

Mapping Java types

Sometimes you need to create pure-Java objects in Jython (for example, when you need to create objects to send across a network to a Java-based server, or when you need to pass the object to a type-sensitive Java service, such as with Swing table cell values). To convert Jython built-in types to Java types (and vice versa) use the functions in the following example (a continuation of the `JavaUtils.py` module excerpt from Wrapping Java services in Jython on page 40):

```
def mapMapFromJava (map):
    """ Convert a Map to a Dictionary. """
    result = {}
    iter = map.keySet().iterator()
    while iter.hasNext():
        key = iter.next()
        result[mapFromJava(key)] = mapFromJava(map.get(key))
    return result

def mapCollectionFromJava (coll):
    """ Convert a Collection to a List. """
    result = []
    iter = coll.iterator();
    while iter.hasNext():
        result.append(mapFromJava(iter.next()))
    return result

def mapFromJava (object):
    """ Convert a Java type to a Jython type. """
    if object is None: return object
    if   isinstance(object, util.Map):
        result = mapMapFromJava(object)
    elif isinstance(object, util.Collection):
        result = mapCollectionFromJava(object)
    else:
        result = object
    return result

def mapSeqToJava (seq):
    """ Convert a sequence to a Java ArrayList. """
    result = util.ArrayList(len(seq))
    for e in seq:
        result.add(mapToJava(e));
    return result

def mapDictToJava (dict):
    """ Convert a Dictionary to a Java HashMap. """
    result = util.HashMap()
    for key, value in dict.items():
        result.put(mapToJava(key), mapToJava(value))
    return result

def mapToJava (object):
    """ Convert a Jython type to a Java type. """
    if object is None: return object
    t = type(object)
    if   t == TupleType or t == ListType:
        result = mapSeqToJava(object)
    elif t == DictType:
        result = mapDictToJava(object)
    else:
        result = object
    return result
```

After using `mapToJava`, these types can be written to a `java.io.ObjectOutputStream`. After reading an object from a `java.io.ObjectInputStream`, you can use `mapFromJava` to convert the object back to a Jython type.

Note that these methods support a limited but broadly used set of built-in Jython types. Jython automatically converts value-like types such as numbers and strings. User defined classes are not supported.

Mapping Java types, continued

To continue the example, the following usage of the mapping functions discussed on the previous panel as shown here:

```
data = (1,2,3, [1,2,3], [c for c in "Hello!"], "Hello!", {1:'one', 2:'two'})
print "data:", data
toJava = mapToJava(data)
print "toJava:", toJava
fromJava = mapFromJava(toJava)
print "fromJava:", fromJava

print

print "type(%s)=%s" % ("data", type(data))
print "type(%s)=%s" % ("toJava", type(toJava))
print "type(%s)=%s" % ("fromJava", type(fromJava))
```

prints:

```
data: (1, 2, 3, [1, 2, 3], ['H', 'e', 'l', 'l', 'o', '!'], 'Hello!', \
    {2: 'two', 1: 'one'})
toJava: [1, 2, 3, [1, 2, 3], [H, e, l, l, o, !], Hello!, {2=two, 1=one}]
fromJava: [1, 2, 3, [1, 2, 3], ['H', 'e', 'l', 'l', 'o', '!'], 'Hello!', \
    {2: 'two', 1: 'one'}]

type(data)=org.python.core.PyTuple
type(toJava)=org.python.core.PyJavaInstance
type(fromJava)=org.python.core.PyList
```

Notice that the `PyTuple` became a `PyJavaInstance` and then a `PyList`. Also notice that the `toJava` form formats differently. This is because it is a Java object and it's being printed by the Java `toString()` method, not Jython `repr()` function. `PyJavaInstance` is a type Jython will pass as is to a Java API. Finally, notice that the `data` and `fromJava` values are the same except that the tuple is now an equivalent list. For more about Jython types see Appendix L: Jython types summary on page 99 .

Parsing command lines

Frequently you need to extract command parameters with more processing than simple use of `sys.argv` provides. The `parseArgs` function can be used to get any command line arguments as a (tuple of) sequence of positional arguments and a dictionary of switches.

So, continuing the `JavaUtils.py` module excerpt (from Wrapping Java services in Jython on page 40 and Mapping Java types on page 41 , respectively), we see this:

```
def parseArgs (args, validNames, nameMap=None):
    """ Do some simple command line parsing. """
    # validNames is a dictionary of valid switch names -
    #   the value (if any) is a conversion function
    switches = {}
    positionals = []
    for arg in args:
        if arg[0] == '-':               # a switch
            text = arg[1:]
            name = text; value = None
            posn = text.find(':')       # any value comes after a :
            if posn >= 0:
                name = text[:posn]
                value = text[posn + 1:]
            if nameMap is not None:     # a map of valid switch names
                name = nameMap.get(name, name)
            if validNames.has_key(name):   # or - if name in validNames:
                mapper = validNames[name]
                if mapper is None: switches[name] = value
                else:              switches[name] = mapper(value)
            else:
                print "Unknown switch ignored -", name

        else:                           # a positional argument
            positionals.append(arg)
    return positionals, switches
```

This function could be used as follows (in file `parsearg.py`):

```
from sys import argv
from JavaUtils import parseArgs

switchDefs = {'s1':None, 's2':int, 's3':float, 's4':int}
args, switches = parseArgs(argv[1:], switchDefs)
print "args:", args
print "switches:", switches
```

For the command `c:\>jython parsearg.py 1 2 3 -s1 -s2:1 ss -s4:2`, it prints:

```
args: ['1', '2', '3', 'ss']
switches: {'s4': 2, 's2': 1, 's1': None}
```

Section 8. Jython string processing

String operations and functions

Like most scripting languages, such as Perl and Rexx, Jython has extensive support for manipulating strings. This support is generally similar to the support provide by the Java language but it is often simpler and easier to use. In this section, we will talk about some of the more commonly used string operations and functions. See Part 1 of this tutorial and the *Python Library Reference* to learn more about string methods.

In the examples in the next few sections I will use the following values:

```
name ="Barry Feigenbaum"
addr = '12345 Any Street"
v1 = 100; v2 = v1 * 1.5; v3 = -v2; v4 = 1 / v2
s1 = "String 1"; s2 = "String 2"
sent = "The rain in Spain falls mainly on the plain."
```

Getting string forms of objects

To get a string representation of any value or expression (that is, object) use one of the following functions:

- **str(expr)** creates a human-oriented string.
- **repr(expr)** or **`expr`** creates (where possible) a computer-oriented string from which the `eval` function can re-create the value.

Note that for many types, including basic types, `str(x)` and `repr(x)` generate the same (or very similar) strings.

Basic string operations

A string is a built-in type, acting both as a value and as an object with methods. Strings support the basic operations of concatenation, indexing, containment, and formatting, as well as the other operations of immutable sequences. We'll go over the basic string operations, starting with concatenation.

We use the plus (+) operator to *concatenate* two strings. For example, the following line:

```
print "abc" + "xyz"
```

prints: `abcxyz`.

To select a character or characters (that is, a substring) from a string you use indexing. For example: `"abcxwy"[2]` yields *c*, while `"abcxwy"[2:4]` yields *cx*.

Many of the string functions test conditions, thus they are often used in conjunction with the `if` and `while` statements. Here's an example of how we could use containment testing to see if a character were contained in a string:

```
if ' ' in name: print "space found"

-- or --

if 'q' not in sent: print "q not found"
```

In addition to testing *conditions*, strings also support methods to test the *nature* of the string. These are `islower`, `isupper`, `isalnum`, `isnum`, `isalpha`, `isspace`, and `istitle`. These methods test to see if all the characters in the strings meet these conditions.

Additional methods

Strings support several methods that allow you to find and edit sub-strings, change case, and a host of other actions. To find a string in another string use the `find`/`rfind` or `startswith`/`endswidth` methods. For example:

```
if name.find(' ') >= 0: print "space found"

-- or --

if name.find("Jones") < 0: print "Jones not in name"
```

Sometimes you need to edit the content of a string, for example to change its case or insert or remove text from it. Jython supplies several methods to do this. To change case, Jython has the `lower`, `upper`, `swapcase`, `title`, and `capitalize` methods. To change the text of a string, use the `replace` method. For example, to match strings often you want to ignore case or you may want to replace sub-strings:

```
if  s1.lower() == s2.lower(): print "equal"

-- or --
```

```
newaddr = addr.replace("Street", "St.")
```

Often strings have extra blanks around them that are not important, such as when the string is entered by a user. To remove these extra blanks use the `lstrip`, `rstrip`, or `strip` methods. For example, to match a command entered by a user:

```
cmd = raw_input("Enter a command")
if cmd.lstrip.startswith("run "):
    print "run command found"
```

Often you need to break strings into parts, such as the words in a sentence or join multiple strings into one string. Jython supports the `split`, `splitlines`, and `join` functions to do this. The `split` method splits a line into words, while `splitlines` splits a file of lines into separate lines. The `join` method reverses `split`. You can also join strings by concatenation as discussed above. For example, to extract the words from a sentence and then rebuild the sentence use:

```
words = sent.split(' ')   # use space to separate words
sent2 = ' '.join(words)   # use space between words
```

Formatting program variables

It is very easy to format local or global variables using the modulus (%) operator. The `locals` and `globals` functions return dictionaries for all the local and global (respectively) variables. For example:

```
fname = "Barry"; lname = "Feigenbaum"
address = "1234 any St."
city = "Anytown"; state = "TX"; zip = "12345"
age = 30
children = 3
  :
print "Hello %(fname)s from %(city)s, %(state)s." % locals()
```

prints `Hello Barry from Anytown, TX.`

See Appendix J: Formatting strings and values on page 94 for more about formatting program variables.

Format operator examples

Below are some format (%) operator examples. See Appendix J: Formatting strings and values on page 94 for more examples.

Expression	Result
`"Hello %s" % "Barry"`	Hello Barry
`"Count: %i, " "Avg Cost: $%.2f; " "Max Cost: $%.2f" % (10, 10.5, 50.25)`	Count: 10, Avg Cost: $10.50; Max Cost: $50.25
`"This is %i%%" % 10`	This is 10%
`"My name is %(first)s %(last)s!" % {'last':'Feigenbaum', 'first':'Barry', 'mi':'A'}`	My name is Barry Feigenbaum!

Using C-style printf

For those familiar with C's `printf("... %x ...", v1, ..., vN)` function, a similar but enhanced service can be added in Jython, as shown here:

```
def printf(stream, format, *pargs, **kwargs):
    # see Printing to files on page 62   for more information
    if   pargs:
        print >>stream, format % pargs
    elif kwargs:
        print >>stream, format % kwargs
    else:
        print >>stream, format
```

Using the above `printf` function definition, the following examples:

```
from sys import stdout

printf(stdout, "%s is %.1f years old and has %i children",
       fname, age, children)

printf(stdout, "The %(name)s building has %(floors)d floors",
       floors=105, name="Empire State")

printf(stdout, "Hello World!")
```

print:

```
Barry is 30.0 years old and has 3 children
```

```
The Empire State building has 105 floors
Hello World!
```

Pretty printing

You can use the `pprint` module functions, in particular the `pformat` function, to print complex data structures in a formatted form. For example, this code:

```
data = [[1,2,3], [4,5,6],{'1':'one', '2':'two'},
        "jsdlkjdlkadlkad", [i for i in xrange(10)]]
print "Unformatted:"; print data

print

from pprint import pformat
print "Formatted:"; print pformat(data)
```

prints the following:

```
Unformatted:
[[1, 2, 3], [4, 5, 6], {'2': 'two', '1': 'one'}, \
    'jsdlkjdlkadlkad', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]

Formatted:
[[1, 2, 3],
 [4, 5, 6],
 {'2': 'two', '1': 'one'},
 'jsdlkjdlkadlkad',
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
```

Using string functions

As an example of using the string operations from String operations and functions on page 46 , the `justify.py` program (listed below) takes paragraphs of input and formats them into pages. The text may be left-, center-, right-aligned, or justified. Page margins may be specified. Header and/or footer text may be supplied.

See Resources on page 73 for some sample results of using this program.

```
import sys

def stripLines (lines):
    """ Removed extra whitespace (that is, newlines). """
    newlines = []
```

```
    for line in lines:
        line = line.strip()
        newlines.append(line)
    return newlines

def splitParagraphs (lines):
    """ Splits a set of lines into paragraphs.  """
    paras = []
    para = ""
    for line in lines:
        if len(line) > 0:       # in paragraph
            para += ' ' + line
        else:                   # between paragraphs
            para = para.strip()
            if len(para) > 0:
                paras.append(para)
                para = ""
    return paras

class Formatter:
    """ Formats and prints paragraphs.  """

    def __init__ (self, stream, pagelen=66, linewidth=85,
                        lmargin=10, rmargin=10, pindent=5,
                        alignment="justify",
                        headers=None, footers=None):
        self.stream = stream        # stream to print on

        # format settings
        self.pagelen = pagelen
        self.pindent = pindent
        self.linewidth = linewidth
        self.lmargin = lmargin
        self.rmargin = rmargin
        self.headers = headers
        self.footers = footers
        self.alignment = alignment

        self.pagecount = 1         # current page
        self.linecount = 0         # current line

    def genLine (self, line):
        print >>self.stream, line
        self.linecount += 1

    def outputLine (self, line):
        self.testEndPage()
        if not (self.linecount == 0 and len(line) == 0):
            self.genLine(line)

    def newPage (self):
        if self.headers:
            self.outputHeader()

    def padPage (self):
        while self.linecount < self.pagelen:
            self.genLine("")
```

```
    def endPage (self):
        if self.footers:
            if len(self.footers) + self.linecount < self.pagelen:
                self.padPage()
            self.outputFooter()
        else:
            if self.linecount < self.pagelen:
                self.padPage()
        self.linecount = 0
        self.pagecount += 1
        self.genLine('-' * 20)

    def testEndPage (self):
        if self.footers:
            if len(self.footers) + 1 + self.linecount >= self.pagelen:
                self.endPage()
                self.newPage()
        else:
            if self.linecount >= self.pagelen:
                self.endPage()
                self.newPage()

    def padLine (self, line, firstline=0, lastline=0):
        """ Add spaces as needed by alignment mode.  """

        if   self.alignment == "left":
            adjust = firstline * self.pindent
            #line = line

        elif self.alignment == "center":
            adjust = 0
            pad = self.linewidth - adjust - len(line)
            line = ' ' * (pad / 2) + line

        elif self.alignment == "right":
            adjust = 0
            pad = self.linewidth - adjust - len(line)
            line = ' ' * pad + line

        elif self.alignment == "justify":
            adjust = firstline * self.pindent
            pad = self.linewidth - adjust - len(line)
            line = ""

            # add 1+ spaces between words to extend line
            words = line.split()
            xpad = pad
            for word in words:
                line += word + ' '
                if not lastline and xpad > 0:
                    line += ' '  * (pad / len(words) + 1)
                    xpad -= 1
            line = line.strip()

        return ' ' * adjust + line

    def format (self, line, firstline=0, lastline=0):
        # indent by left margin
```

```
        return ' ' * self.lmargin + \
               self.padLine(line.strip(), firstline, lastline)

    def formatParagraph (self, para):
        lcount = 0
        adjust = self.pindent
        line = ""

        # process by words
        words = para.split(' ')
        for word in words:
            line += ' '
            # about to get too long
            if len(line) + len(word) > self.linewidth - adjust:
                line = self.format(line, lcount == 0, 0)
                self.outputLine(line)
                line = ""
                lcount += 1
                adjust = 0
            line += word
        # output last (only) line
        if len(line) > 0:
            line = self.format(line, lcount == 0, 1)
            self.outputLine(line)

    def outputHeader (self):
        for line in self.headers:
            self.genLine(' ' * self.lmargin + line.center(self.linewidth))
        self.genLine("")

    def outputFooter (self):
        self.genLine("")
        for line in self.footers:
            self.genLine(' ' * self.lmargin + line.center(self.linewidth))

    def outputPages (self, paras):
        """ Format and print the paragraphs. """
        self.newPage()
        for para in paras:
            self.formatParagraph(para)
            self.outputLine("")
        self.endPage()
```

Section 9. Processing regular expressions

About regular expressions

As an extension to the find and replace functions described in String operations and functions on page 46 , Jython supports regular expressions. *Regular expressions* (RE) are strings that contain plain match text and control characters and provide an extremely powerful string search and replace facility. Jython supports (at least) the following forms of regular expressions:

- **re module** is a built-in part of Jython.
- **Java** works if you're running Jython on Java 1.4 or above.
- **Apache ORO** works if you add the `ORO` package to your `CLASSPATH`.

Regular expression formats

The simplest RE is an exact string to match. More complex REs include special control characters. The control characters allow you to create patterns of matching strings. For more information on RE syntax and options see Appendix H: Regular expression control characters on page 84 and the *Python Library Reference*.

Below are some example REs and the strings they match:

Control character	Regular expression	Matches	Does not match
-- none --	abc	abc	ab aabc abcc
. - any character	a.c	abc axc a c	ac abbc
* - optional repeating subpattern	a.*c	abc axc a c ac	abcd

		axxxxc	
? - optional subpattern	a.?c	abc	ac aabc
+ - required repeating subpattern	a.+c	abc abbc axxc	ac abcd
...|... - choice of subpattern	abc|def	abcef abdef	abef abcdef
(...) - grouping	a(xx)|(yy)c	axxc ayyc axxyyc	axc ayc
(...)* - repeating grouping	a(xx)*c	ac axxc axxxxc	axxbxxc
(...)+ - required repeating grouping	a(xx)+c	axxc axxxxc	ac axxbxxc
\c - match a special character	\.\?*\+	.?*+	?.*+ abcd
\s - matches white space	a\s*z	az a z a z	za z a abyz

Regular expressions functions

The Jython `re` module provides support for regular expressions. `re`'s primary functions are `findall`, `match`, and `search` to find strings, and `sub` and `subn` to edit them. The `match` function looks at the start of a string, the `search` function looks anywhere in a string, and the `findall` function repeats `search` for each possible match in the string. `search` is (by far) the most used of the regular expression functions.

Here are some of the most common RE functions:

Function	Comment(s)

`match(pattern, string {, options})`	Matches pattern at the string start
`search(pattern, string {, options})`	Matches pattern somewhere in the string
`findall(pattern, string)`	Matches all occurrences of pattern in the string
`split(pattern, string {, max})`	Splits the string at matching points and returns the results in a list
`sub(pattern, repl, string {, max})`	Substitutes the match with repl for max or all occurrences; returns the result
`subn(pattern, repl, string {, max})`	Substitutes the match with repl for max or all occurrences; returns the tuple (result, count)

Note that the `match`ing functions return `None` if no match is found. Otherwise the `match` functions will return a `Match` object from which details of the match can be found. See the *Python Library Reference* for more information on `Match` objects.

Two function examples

Let's take a look at some examples of regular expressions functions in action:

```
import re

# do a fancy string match
if re.search(r"^\s*barry\s+feigenbaum\s*$", name, re.I):
   print "It's Barry alright"

# replace the first name with an initial
name2 = re.sub(r"(B|b)arry", "B.", name)
```

If you are going to use the same pattern repeatedly, such as in a loop, you can speed up execution by using the `compile` function to compile the regular expression into a `Pattern` object and then using that object's methods, as shown here:

```
import re
patstr = r"\s*abc\s*"
pat = re.compile(patstr)
# print all lines matching patstr
for s in stringList:
    if pat.match(s, re.I): print "%r matches %r" % (s, patstr)
```

Regular expression example: Grep

The following simplified version of the `Grep` utility (from `grep.py`) offers a more complete example of a Jython string function.

```
""" A simplified form of Grep. """

import sys, re

if len(sys.argv) != 3:
    print "Usage: jython grep.py <pattern> <file>"
else:
    # process the arguments
    pgm, patstr, filestr = sys.argv
    print "Grep - pattern: %r file: %s" % (patstr, filestr)
    pat = re.compile(patstr)  # prepare the pattern

    # see File I/O in Jython on page 58   for more information
    file = open(filestr)      # access file for read
    lines = file.readlines()  # get the file
    file.close()

    count = 0
    # process each line
    for line in lines:
        match = pat.search(line)    # try a match
        if match:                   # got a match
            print line
            print "Matching groups: " + str(match.groups())
            count += 1
    print "%i match(es)" % count
```

When run on the words.txt file from File I/O in Jython on page 58 , the program produces the following result:

```
C:\Articles>jython grep.py "(\w*)!" words.txt
Grep - pattern: '(\\w*)!' file: words.txt
How many times must I say it; Again! again! and again!

Matched on: ('Again',)
Singing in the rain! I'm singing in the rain! \
    Just singing, just singing, in the rain!

Matched on: ('rain',)
2 match(es)
```

Section 10. File I/O in Jython

Using files

In addition to the Java platform's file-related APIs (packages `java.io` and, in Java 1.4, `java.nio`), Jython provides simple yet powerful access to files using the `File` type and services in the `os`, `os.path`, and `sys` modules. (See Appendix F: The os module on page 83, Appendix G: The os.path module on page 84, Appendix E: The sys module on page 82 and the *Python Reference Manual* for more details on the `os` and `os.path` packages.)

We'll start with a look at some basic file-access operations. A `File` object is created using the built-in `open` function, shown below, where `path` is the path to the file, `mode` is the access mode string, and `size` is the suggested buffer size:

```
file = open(path {, mode {, size}})
```

The `mode` string has the following syntax: `(r|w|a){+}{b}`; the default mode is `r`. Here is a listing of all the available access mode strings:

- **r**: read
- **w**: write
- **a**: append to the end of the file
- **+**: update
- **b**: binary (vs. text)

The name of the file is accessed through the `name` attribute. The mode of the file is accessed through the `mode` attribute.

File access methods

Files support the following methods:

Method	Comment(s)
`close()`	Flush and close an open file
`flush()`	Outputs any buffered data
`read({size})`	Reads up to size (or the whole file)
`readline({size})`	Read a line (including ending '\n') up to size
`readlines()`	Reads the file and returns a list of lines (including

	ending '\n')
`seek(offset {, mode})`	Seek to a position, mode: 0 - start of file, 1 - current offset, 2 - end of file
`tell()`	Return the current offset
`truncate({size})`	Truncate (delete extra content) to current offset or specified size
`write(string)`	Write the string to a file. To write lines, end the string in '\n'
`writelines(lines)`	Write the list as a set of strings. To write lines, end each string in '\n'

Simple file processing examples

We'll look at a couple of simple file processing examples, starting with the file copy program below:

```
import sys

f = open(sys.argv[1], "rb")    # open binary for reading
bin = f.read()
f.close()
f = open(sys.argv[2], "wb")    # open binary (truncated) for write
f.write(bin)
f.close()
```

And here is a text file sort procedure:

```
import sys

f = open(sys.argv[1], "r")    # read the file by lines
lines = f.readlines()
f.close()
lines.sort()                  # sort and print the lines
print "File %s sorted" % f.name
print lines
```

A word-counting program in Jython

As a more complete example of file processing, study the following word-counting program:

```
import sys

def clean (word):
    """ Remove any punctuation and map to a common case. """
    word = word.lower()
    # remove any special characters
    while word and word[-1] in ".,;!": word = word[:-1]
    while word and word[0] in ".,;!": word = word[1:]
    return word

words = {}  # set of unique words and counts

if len(sys.argv) != 2:
    print "Usage: jython wcount.py <file>"
else:
    file = open(sys.argv[1])  # access file for read
    lines = file.readlines()  # get the file
    file.close()

    # process each line
    for line in lines:
        # process each word in the line
        for word in line.split():
            word = clean(word)
            words[word] = words.get(word, 0) + 1   # update the count

    # report the results
    keys = words.keys()
    keys.sort()
    for word in keys:
        print "%-5i %s" % (words[word], word)
```

Output of words.txt

Given the following input file (*words.txt*)

```
Now is the time for all good men to come to the aid of their country.
The rain in Spain falls mainly on the plain.
How many times must I say it; Again! again! and again!
Singing in the rain! I'm singing in the rain! \
    Just singing, just singing, in the rain!
```

the word-counting program (from A word-counting program in Jython on page 59) would return the following results (wrapped into two columns to save space):

```
3     again             1     many
1     aid               1     men
1     all               1     must
1     and               1     now
1     come              1     of
```

```
1     country          1     on
1     falls            1     plain
1     for              4     rain
1     good             1     say
1     how              4     singing
1     i                1     spain
1     i'm              7     the
4     in               1     their
1     is               1     time
1     it               1     times
2     just             2     to
1     mainly
```

The word-counting program in Java code

Let's take a look at the word-counting script re-implemented in the Java language. Notice the extensive use of types in declarations and type-casts in the assignment statements. As you can see, the Java code is significantly larger (approximately two times) and arguably far more cryptic.

```
import java.io.*;
import java.util.*;
import java.text.*;

public class WordCounter
{
    protected static final String specials = ".,;!";

    /** Remove any punctuation and map to a common case. */
    protected static String clean(String word) {
        word = word.toLowerCase();
        // remove any special characters
        while (word.length() > 0 &&
               specials.indexOf(word.charAt(word.length() - 1)) >= 0) {
            word = word.substring(0, word.length() - 1);

        }
        while (word.length() > 0 &&
               specials.indexOf(word.charAt(0)) >= 0) {
            word = word.substring(1);
        }
        return word;
    }

    protected static Map words = new HashMap();

    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.out.println("Usage: java WordCounter <file>");
        }
        else {
            // access file for read
```

```
            FileInputStream fis = new FileInputStream(args[0]);
            DataInputStream dis = new DataInputStream(fis);
            List lines = new ArrayList();
            // get the file
            for (String line = dis.readLine();
                 line != null;
                 line = dis.readLine()) {
                lines.add(line);
            }
            dis.close();

            // process each line
            for (int i = 0; i < lines.size(); i++) {
                String line = (String)lines.get(i);
                System.out.println("Processing: " + line);
                String[] xwords = line.split("\\s+");
                for (int w = 0; w < xwords.length; w++) {
                    String word = clean(xwords[w]);
                    if (word.length() > 0) {
                        Integer count = (Integer)words.get(word);
                        if (count == null) {
                            count = new Integer(0);
                        }
                        // update the count
                        words.put(word,
                                  new Integer(count.intValue() + 1));
                    }
                }
            }

            // report the results
            String[] keys = (String[])words.keySet().
                                toArray(new String[words.size()]);
            Arrays.sort(keys);

            MessageFormat form = new MessageFormat(
                              "{0,number, #########0} {1}");
            for (int i = 0; i < keys.length; i++) {
                System.out.println(form.format(
                           new Object[] {words.get(keys[i]), keys[i]}));
            }
        }
    }
}
```

Printing to files

The `print` statement can print to a file by use of the ">>" operator. By default it prints to the console (actually the value of `sys.stdout`). For example, the following commands are equivalent:

```
print "Hello World!"
```

```
import sys
print >>sys.stdout, "Hello world!"
```

Jython allows alternate target files. For example, to print to the standard error stream use:

```
print >>sys.stderr, "Hello world!"
```

To print to a file use:

```
f = open("myfile", "w")
for i in range(10):
    print >>f, "Line", i
f.close()
```

And to add to the end of a file use:

```
f = open("myfile", "a")
print >>f, "Added line"
f.close()
```

Saving objects persistently

Sometimes you may want to save an object persistently (beyond the lifetime of the program that creates it) or send it to another application. To do this you need to *serialize* (or *pickle*) the object so it can be placed in a file or on a stream. You then need to *de-serialize* (or *un-pickle*) the object to use it again. Jython provides a module, `pickle`, that makes this very easy. The `pickle` module contains the following useful functions:

Function	**Comment(s)**
`load(file)`	Returns an object re-created from a previously created image in a file.
`loads(string)`	Returns an object recreated from a previously created image in a string.
`dump(object, file {, bin})`	Stores an object image into a file. If `bin` is omitted or false, use a text representation; else a binary representation (which is typically smaller).
`dumps(object{, bin})`	Returns a string containing the image of the object. If `bin` is omitted or false, use a text representation; else a binary representation

	(which is typically smaller).

A pickling example

Here's an example of `pickle` at work. The following code sequence

```
import pickle

class Data:
    def __init__ (self, x, y):
        self.__x = x
        self.__y = y

    def __str__ (self):
        return "Data(%s,%s)" % (self.__x, self.__y)

    def __eq__ (self, other):
        return self.__x == other.__x and self.__y == other.__y


data = Data(10, "hello")

file = open("data.pic", 'w')
pickle.dump(data, file)
file.close()

file = open("data.pic", 'r')
newdata = pickle.load(file)
file.close()

print "data:", data
print "newdata:", newdata
print "data is newdata:", data is newdata
print "data == newdata:", data == newdata
```

prints this:

```
data: Data(10,hello)
newdata: Data(10,hello)
data is newdata: 0 (false)
data == newdata: 1 (true)
```

The file created is in (semi-)readable plain text. For example, the above code created the file `data.pic`:

```
(i__main__
Data
p0
```

```
(dp1
S'_Data__y'
p2
S'hello'
p3
sS'_Data__x'
p4
I10
sb.
```

Note that Jython cannot pickle objects that are Java objects, reference Java objects, or subclass Java classes. To do this you need to use the `java.io.ObjectOutputStream` and `java.io.ObjectInputStream` classes.

Object shelves

As shown in the previous panel, Jython can store objects into a file. Using a file per object can cause problems (that is, it can waste space and you will need to name each file). Jython supports a file that can hold multiple objects, called a *shelf*. A shelf acts much like a persistent dictionary. To create shelves, use the `open` function of module `shelve`. For example, the following code:

```
import shelve, sys

def printshelf (shelf, stream=sys.stdout):  # print the entries in a shelf
    for k in shelf.keys():
        print >>stream, k, '=', shelf[k]

def clearshelf (shelf):                     # remove all keys in the shelf
    for k in shelf.keys():
        del shelf[k]

# create shelf
shelf = shelve.open("test.shelf")
clearshelf(shelf)
shelf["x"] = [1,2,3,4]
shelf["y"] = {'a':1, 'b':2, 'c':3}
printshelf(shelf)
shelf.close()

print
# update shelf
shelf = shelve.open("test.shelf")
printshelf(shelf)
print
shelf["z"] = sys.argv[1]
printshelf(shelf)
shelf.close()

print
# verify shelf persistent
```

```
shelf = shelve.open("test.shelf")
printshelf(shelf)
shelf.close()
```

produces this output (with argument "This is a test string"):

```
x = [1, 2, 3, 4]
y = {'b': 2, 'a': 1, 'c': 3}

x = [1, 2, 3, 4]
y = {'b': 2, 'a': 1, 'c': 3}

x = [1, 2, 3, 4]
z = This is a test string
y = {'b': 2, 'a': 1, 'c': 3}

x = [1, 2, 3, 4]
z = This is a test string
y = {'b': 2, 'a': 1, 'c': 3}
```

Note that the `open` function produces two files based on the file name passed to `open`:

- **<filename>.dir** is a directory into the persistent data
- **<filename>.dat** is the saved persistent object data

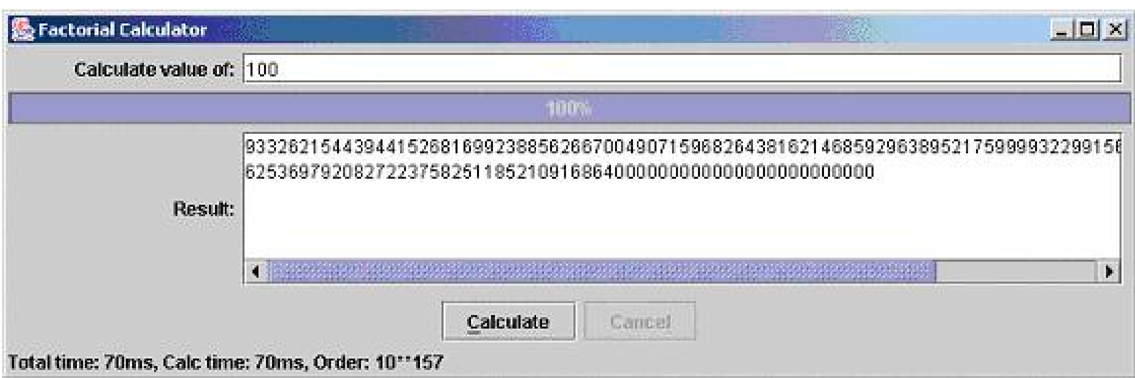
Section 11. A simple Swing GUI

The Factorial Calculator

We'll close this second installment of the "Introduction to Jython" tutorial with a complete program that encompasses many of the details we have so far discussed. The Factorial Calculator is a GUI application written entirely in Jython. It calculates the value of *x!* (x factorial) for any positive integer value. Because *x!* can be very large, this example takes advantage of Jython's ability to process very large integers. Calculations for large values of x (say, > 10000) can take several minutes, so the user interface includes a progress bar and a Cancel button to interrupt a calculation.

In the panels that follow, you can see the two most essential components of the Factorial Calculator: the class that supplies the factorial calculation engine, and the set of classes that comprise the GUI. The complete, runnable code for the Factorial Calculator is available for download in Resources on page 73 . Note that in order to completely understand the GUI code you should have some experience with creating Swing GUIs. Even without this prior knowledge, you should be able to discern many elements of the code from our prior discussion throughout this tutorial.

To get started, let's see what our GUI application looks like. Here's a screenshot of the GUI showing the result of calculating 100! (that is, 100 factorial).



The factorial engine: factor.py

`Factorial` is the class that supplies the factorial calculation engine. It consists of a sequence of code with additional explanation lines (identified by --) added.

```
-- import the needed modules
import sys
import exceptions

-- create the Factorial class, a general purpose factorial calculation engine
class Factorial:
    """A general purpose factorial calculation engine"""

-- define the constructor
    def __init__ (self):
        self.__listeners = []
        self.__cancelled = 0

-- allow other classes to register event listeners;
---    used to track calculation progress
-- A "listener" is a function that takes an integer % argument
    def addListener (self, listener):
        if listener not in self.__listeners:
            self.__listeners.append(listener)

    def addListeners (self, listeners):
        for l in listeners:
            self.addListener(l)

    def removeListener (self, listener):
        self.__listeners.remove(listener)

    def removeListeners (self, listeners):
        for l in listeners:
            self.removeListener(l)

    def fireListeners (self, value):  # notify all listeners
        for func in self.__listeners:
            func(value)

-- allow others to cancel a long running calculation
    def cancel (self):
        self.__cancelled = 1

-- perform the factorial calculation;
--     may take a long time (many minutes) for big numbers
    def calculate (self, value):
        if type(value) != type(0) or value < 0:
            raise ValueError,  \
               "only positive integers supported: " + str(value)

        self.__cancelled = 0
        result = 1L
        self.fireListeners(0)   # 0% done
        # calculate factorial -- may take quite a while
        if value > 1:           # need to do calculation
            last = 0
            # using iteration (vs. recursion) to increase performance
            # and eliminate any stack overflow possibility
            for x in xrange(1, value + 1):
                if self.__cancelled: break  # early abort requested
                result = result * x         # calc next value
                next = x * 100 / value
```

```
                if next != last:            # signal progress
                    self.fireListeners(next)
                    last = next
        self.fireListeners(100)  # 100% done
        if self.__cancelled: result = -1
        return result

# test case
if __name__ == "__main__":
    print sys.argv[0], "running..."
    fac = Factorial()

    def doFac (value):
        try:
            print "For", value, "result =", fac.calculate(value)
        except ValueError, e:
            print "Exception -", e

    doFac(-1)
    doFac(0)
    doFac(1)
    doFac(10)
    doFac(100)
    doFac(1000)
```

The GUI: fgui.py

Here you can see the set of classes that supplies the factorial GUI. The set consists of a sequence of code with additional explanation lines (identified by --) added.

```
-- import the needed modules
import sys
import string
from types import *

from java import lang
from java import awt
from java.awt import event as awtevent
from javax import swing

from factor import Factorial

-- PromptedValueLayout is a customized Java LayoutManager not discussed here
--    but included with the resources
from com.ibm.articles import PromptedValueLayout as ValueLayout

-- support asynchronous processing
class LongRunningTask(lang.Thread):
    def __init__ (self, runner, param=None):
        self.__runner = runner   # function to run
        self.__param = param     # function parameter (if any)
        self.complete = 0
```

```
        self.running = 0

-- Java thread body
    def run (self):
        self.complete = 0; self.running = 1
        if self.__param is not None:
            self.result = self.__runner(self.__param)
        else:
            self.result = self.__runner()
        self.complete = 1; self.running = 0

-- start a long running activity
def doAsync (func, param):
    LongRunningTask(func, param).start()

-- Swing GUI services must be called only on the AWT event thread,
class SwingNotifier(lang.Runnable):
    def __init__ (self, processor, param=None):
        self.__runner = processor  # function to do GUI updates
        self.__param = param       # function parameter (if any)

-- Java thread body
    def run (self):
        if self.__param is not None: self.__runner(self.__param)
        else:                        self.__runner()

    def execute (self):
        swing.SwingUtilities.invokeLater(self)

-- define and construct a GUI for factorial calculation
class FactorialGui(swing.JPanel):
    """Create and process the GUI."""

    def __init__ (self, engine):
        swing.JPanel.__init__(self)
        self.__engine = engine
        engine.addListener(self.update)
        self.createGui()

    def update (self, value):          # do on AWT thread
        SwingNotifier(self.updateProgress, value).execute()

    def updateProgress (self, value):  # display progress updates
        self.__progressBar.value = value

-- Calculate button press handler
    def doCalc (self, event):          # request a factorial
        self.__outputArea.text = ""
        ivalue = self.__inputField.text # get value to calculate
        value = -1
        try: value = int(ivalue)        # convert it
        except: pass
        if value < 0:                   # verify it
           self.__statusLabel.text = \
               "Cannot make into a positive integer value: " + ivalue
        else:
            self.__calcButton.enabled = 0
            self.__cancelButton.enabled = 1
```

```
            msg = "Calculating factorial of %i..." % value
            if value > 25000: msg += \
                 "; May take a very long time to complete!"
            self.__statusLabel.text = msg  # tell user we're busy
            doAsync(self.calcFac, value)   # do the calculation

-- main calculation worker
    def calcFac (self, value):
        stime = lang.System.currentTimeMillis()
        fac = self.__engine.calculate(value)   # time calculation
        etime = lang.System.currentTimeMillis()
        svalue = ""; order = 0
        if fac >= 0:          # we have a result, not cancelled
            svalue = str(fac); order = len(svalue) - 1
            formatted = ""
            while len(svalue) > 100:  # wrap long numbers
                formatted += svalue[0:100] + '\n'
                svalue = svalue[100:]
            formatted += svalue
            svalue = formatted
        ftime = lang.System.currentTimeMillis()

        SwingNotifier(self.setResult, \
          (svalue, order, ftime - stime, etime - stime)).execute()

-- display the result
    def setResult (self, values):
        svalue, order, ttime, ftime = values
        self.__cancelButton.enabled = 0
        if len(svalue) > 0:
            self.__statusLabel.text = \
               "Setting result - Order: 10**%i" % order
            self.__outputArea.text = svalue
            self.__statusLabel.text = \
                "Total time: %ims, Calc time: %ims, Order: 10**%i" % \
                      (ttime, ftime, order)
        else:
            self.__statusLabel.text = "Cancelled"

        self.__calcButton.enabled = 1

-- Cancel button press handler
    def doCancel (self, event):       # request a cancel
        self.__cancelButton.enabled = 0
        self.__engine.cancel()

-- create (layout) the GUI
    def createGui (self):             # build the GUI
        self.layout = awt.BorderLayout()

        progB = self.__progressBar = \
            swing.JProgressBar(0, 100, stringPainted=1);

        inf = self.__inputField = swing.JTextField(5)
        inl = swing.JLabel("Calculate value of:", swing.JLabel.RIGHT)
        inl.labelFor = inf

        outf = self.__outputArea = swing.JTextArea()
```

```
        outl = swing.JLabel("Result:", swing.JLabel.RIGHT)
        outl.labelFor = outf

        calcb = self.__calcButton = \
            swing.JButton("Calculate", actionPerformed=self.doCalc,
                          enabled=1, mnemonic=awtevent.KeyEvent.VK_C)
        cancelb = self.__cancelButton = \
             swing.JButton("Cancel", actionPerformed=self.doCancel,
                          enabled=0, mnemonic=awtevent.KeyEvent.VK_L)

        vl = ValueLayout(5, 5)
        inp = swing.JPanel(vl)
        vl.setLayoutAlignmentX(inp, 0.2)
        inp.add(inl); inp.add(inf, inl)
        self.add(inp, awt.BorderLayout.NORTH)

        vl = ValueLayout(5, 5)
        outp = swing.JPanel(vl)
        vl.setLayoutAlignmentX(outp, 0.2)
        outp.add(outl); outp.add(swing.JScrollPane(outf), outl)

        xoutp = swing.JPanel(awt.BorderLayout())
        xoutp.add(progB, awt.BorderLayout.NORTH)
        xoutp.add(outp, awt.BorderLayout.CENTER)

        self.add(xoutp, awt.BorderLayout.CENTER)

        sp = swing.JPanel(awt.BorderLayout())

        bp = swing.JPanel()
        bp.add(calcb)
        bp.add(cancelb)
        sp.add(bp, awt.BorderLayout.NORTH)

        sl = self.__statusLabel = swing.JLabel(" ")
        sp.add(sl, awt.BorderLayout.SOUTH)
        self.add(sp, awt.BorderLayout.SOUTH)

-- main entry point; launches the GUI in a frame
if __name__ == "__main__":
    print sys.argv[0], "running..."
    frame = swing.JFrame("Factorial Calculator",
                defaultCloseOperation=swing.JFrame.EXIT_ON_CLOSE)
    cp = frame.contentPane
    cp.layout = awt.BorderLayout()
    cp.add( FactorialGui(Factorial()) )
    frame.size = 900, 500
    frame.visible = 1
```

Section 12. Wrap-up and resources

Summary

This completes the two-part "Introduction to Jython" tutorial. While much of the tutorial functions as an overview, I hope I have provided you with enough advanced discussion, code examples, and incentive to proceed into more hands-on learning, specifically by developing your own programs in Jython.

In my opinion, Jython does for the Java platform what Visual Basic does for Microsoft's .NET: It provides much easier access to a complex development and execution environment. While easy to use, Jython improves upon the Java language by incorporating features the Java language lacks (some of which are also available today in .NET languages such as C#) without sacrificing any of the Java platform's capability (unless you count compile-time-type checking or a small reduction in effective performance).

We've discussed many of Jython's enhancements in this tutorial -- including `for each` iteration, property methods accessible as attributes, collection literals, generic collections that hold basic types (such as integers), generic functions, first-class functions, overloadable operators, C-like `printf` formatting, functions as event handlers, and dynamic code execution. Some of these features are so compelling that they will be included in the next version of the Java platform (that is, 1.5). Of course, with Jython you don't have to wait -- you can begin using them today!

Resources

- Download the jython2-source.zip for this tutorial.
- Visit the *Jython home page* to download Jython.
- Take the first part of this tutorial "*Introduction to Jython, Part 1: Java programming made easier*" (*developerWorks*, April 2004).
- Jython modules and packages enable reuse of the extensive standard Java libraries. Learn more about the Java libraries (and download the current version of the JDK) on the Sun Microsystems *Java technology homepage*.
- You'll find an entire collection of Python docs and tutorials (including the *Python Library Reference*) and more information about regular expressions on the *Python*

home page.

* You can also learn more about regular expressions from the tutorial "*Using regular expressions*" (*developerWorks*, September 2000).
* Greg Travis's "*Getting started with NIO*" (*developerWorks*, July 2003) is a good, hands-on introduction to the Java platform's new I/O.
* In "*Charming Jython*" (*developerWorks*, May 2003) regular developerWorks contributor Uche Ogbuji offers a short introduction to Jython.
* Try your hand at using Jython to build a read-eval-print-loop, with Eric Allen's "*Repls provide interactive evaluation*" (*developerWorks*, March 2002).
* *Charming Python* is regular *developerWorks* column devoted to programming with Python.
* Jeffrey Friedl's *Mastering Regular Expressions, Second Edition* (O'Reilly, July 2002) is a comprehensive introduction to regular expressions.
* For a solid introduction to Jython, see Samuele Pedroni and Noel Rappin's *Jython Essentials* (O'Reilly, March 2002).
* *Jython for Java Programmers* focuses on application development, deployment, and optimization with Jython (Robert W. Bill, New Riders, December 2001).
* *Python Programming with the Java Class Libraries* is a good introduction to building Web and enterprise applications with Jython (Richard Hightower, Addison Wesley, 2003).
* You'll find articles about every aspect of Java programming in the *developerWorks Java technology zone*.
* Visit the *Developer Bookstore* for a comprehensive listing of technical books, including hundreds of *Java-related titles* .
* Also see the *Java technology zone tutorials page* for a complete listing of free Java-focused tutorials from *developerWorks*.

Feedback

Please send us your feedback on this tutorial!

Section 13. Appendices

Appendix A: Character codes for array types

The table below lists the character codes for Jython array types (see Java arrays from Jython on page 32).

Character type code	Corresponding Java type
'z'	Boolean
'c'	char
'b'	byte
'h'	short
'i'	int
'l'	long
'f'	float
'd'	double

Note: The above table is from *www.jython.org*.

Appendix B: Common overloaded operators and methods

The following are the operators that are commonly (additional operators can be) overloaded:

Operator	Function to override	Comment(s)
x + y x += y +x	__add__(self, other) __radd__ (self, other) __iadd__(self, other) __pos__ self)	Implements + operator
x - y x -= y -x	__sub__(self, other) __rsub__(self, other) __isub__(self, other)	Implements - operator

	__neg__(self)	
x * y x *= y	__mul__(self, other) __rmul__(self, other) __imul__(self, other)	Implements * operator
x / y x /= y	__div__(self, other) __rdiv__(self, other) __idiv__(self, other)	Implements / operator
x % y x %= y	__mod__(self, other) __rmod__(self, other) __imod__(self, other)	Implements % operator
x & y x &= y	__and__(self, other) __rand__(self, other) __iand__(self, other)	Implements & operator
x | y x |= y	__or__(self, other) __ror__(self, other) __ior__(self, other)	Implements | operator
x ^ y x ^= y	__xor__(self, other) __rxor__(self, other) __ixor__(self, other)	Implements ^ operator
~ x	__invert__(self)	Implements ~ operator
x << y x <<= y	__lshift__(self, other) __rlshift__(self, other) __ilshift__(self, other)	Implements << operator
x >> y x >>= y	__rshift__(self, other) __ rrshift__(self, other) __ irshift__(self, other)	Implements >> operator
x ** y x **= y	__pow__(self, other) __rpow__(self, other) __ipow__(self, other)	Implements ** operator
divmod(x,y)	__divmod__(self, other) __rdivmod__(self, other)	Implements `divmod()`

x < y	__lt__(self, other)	Implements < operator. This should return the opposite value returned by __ge__.
x <= y	__le__(self, other)	Implements <= operator. This should return the opposite value returned by __gt__.
x > y	__gt__(self, other)	Implements > operator. This should return the opposite value returned by __le__.
x >= y	__ge__(self, other)	Implements >= operator. This should return the opposite value returned by __lt__.
x == y	__eq__(self, other)	Implements == operator. This should return the opposite value returned by __ne__.
x != y x <> y	__ne__(self, other)	Implements != operator. This should return the opposite value returned by __eq__.
cmp(x,y)	__cmp__(self, other)	Implements `cmp()`; also used for relational tests if above specific overrides are not defined. This should return a < 0, 0 or > 0 value (say -1, 0 or 1).
x	__nonzero__(self)	Implements logical tests. This should return 0 or 1.
hash(x)	__hash__(self)	Implements `hash()`. Returns an integer value. Instances that are equal (that is, __eq__ returns true) should return the same __hash__ value (that is, `(x == y) and (hash(x) == hash(y))` should be true. Similar to `java.lang.Object.hashCode()`.
abs(x)	__abs__(self)	Implements `abs()`
int(x)	__int__(self)	Implements `int()`
long(x)	__long__(self)	Implements `long()`
float(x)	__float__(self)	Implements `float()`
complex(x)	__complex__(self)	Implements `complex()`
oct(x)	__oct__(self)	Implements `oct()`
hex(x)	__hex__(self)	Implements `hex()`
coerce(x,y)	__coerce__(self, other)	Implements `coerce()`

y = x.name	__getattr__ (self, name)	Implements attribute lookup
x.name = y	__setattr__ (self, name)	Implements attribute creation/update
del x.name	__delattr__ (self, name)	Implements attribute removal
y = c[i]	__getitem_ (self, i)	Implements item lookup
c[i] = y	__setitem__ (self, i)	Implements item creation/update
del c[i]	__delitem__ (self, i)	Implements item removal
x(arg, ...)	__call__ (self, arg, ...)	Implements the *call* operator
len(c)	__len__ (self)	Implements `len()`
x in c x not in c	__contains__ (self, other)	Implements `in` operator
class()	__init__ (self, ...)	Instance constructor; called when the class is created
del x	__del__ (self)	Instance destructor; called just before being deallocated
repr(x) -- or -- `x`	__repr__(self)	Implements `repr()` on this class
str(x)	__str__(self)	Implements `str()` on this class; Jython uses __repr__ if __str__ is not defined. `str()` is used like `x.toString()` in Java

Note: For the binary operators, the __*xxx*__ form is used when the left (or both) argument implements the function; the __***r****xxx*__ form is used only if the right argument implements the function and the left argument does not; the __***i****xxx*__ form is used to implement the augmented assignment (`x ?= y`) operation. See the *Python Reference Manual* for more details and overload-able functions.

Appendix C: Jython debugger commands

The debugger provides the following functions/features:

Command	**Arguments**	**Function**
h, help	-- none --	List the available commands
w, where	-- none --	Shows the current stack trace

d, down	-- none --	Move down one stack frame
u, up	-- none --	Move up one stack frame
b, break	line# | function, condition_expr	Set a breakpoint at a line number or function with an optional expression to evaluate - stop only if true
tbreak	line# | function, condition_expr	Set a breakpoint at a line number or function with an optional expression to evaluate - stop only if true; the breakpoint is automatically cleared when hit
cl, clear	bpid...	Clears all or listed breakpoints
enable	bpid...	Enables breakpoints
disable	bpid...	Disabled breakpoints
ignore	bpid, count	Sets the breakpoint ignore (auto-resume) count
condition	bpid, condition_expr	Sets the breakpoint condition expression
s, step	-- none --	Steps over the next line, possibly into a function
n, next	-- none --	Resume until the next line is reached
r, return	-- none --	Resume until the current function returns
c, cont, continue	-- none --	Resume execution
j, jump	line#	Set a new current line
l, list	line#1, line#1	Lists source from line#1 to line#2, if omitted, then list the lines around the current line
a, args	-- none --	Show the arguments of the current function
p, pp	expr	Evaluate the expression and print its result; *pp* formats the result
print	expr	Do the print statement, that is, - !print expr
alias	name, expr	Create a named expression to simplify printing of repeated values
unalias	name	Delete an alias
q, quit	-- none --	End the debugging session

!	statement	Execute the Jython statement

Note: entering a blank line repeats the prior command.

Appendix D: Jython to/from Java type mapping

Jython uses these rules to map parameter types:

Java Parameter Types	**Allowed Python Types**
char	String (must have length 1)
Boolean	Integer (true = nonzero)
byte, short, int, long	Integer
float, double	Float
java.lang.String, byte[], char[]	String
java.lang.Class	Class or JavaClass
Foobar[]	Array (must contain objects of class or subclass of Foobar)
java.lang.Object	String->java.lang.String, all others unchanged
org.python.core.PyObject	All unchanged
Foobar	Instance --> Foobar (if Instance is subclass of Foobar); JavaInstance --> Foobar (if JavaInstance is instance of Foobar or subclass)

Jython uses these rules to map return value types:

Java Return Type	**Returned Python Type**
char	String (of length 1)
Boolean	Integer (true = 1, false = 0)
byte, short, int, long	Integer
float, double	Float
java.lang.String	String
java.lang.Class	JavaClass which represents given Java class

Foobar[]	Array (containing objects of class or subclass of Foobar)
org.python.core.PyObject (or subclass)	Unchanged
Foobar	JavaInstance which represents the Java Class Foobar

Note: the above two tables are from the *www.jython.org* site.

Appendix E: The sys module

The sys module has some important variables:

Variable	**Comment(s)**
argv	The arguments supplied to the main module. argv[0] is the program name, `argv[1]` is the first argument and so on
maxint minint	Largest/smallest integer value
platform	The version of Java Jython is running on
path	The module search path
stdin stdout stderr	Standard input, output and error streams
modules	List of currently loaded modules
version version_info	Jython version and details

The sys module has some important functions:

Function	**Comment(s)**
exit(int)	Exits the program
exc_info()	Get information on the most recent exception

See the *Python Library Reference* for more information.

Appendix F: The os module

The os module has some important variables:

Variable	**Comment(s)**
name	Type of host
curdir	String to represent the current directory
pardir	String to represent the parent directory
sep	String to separate directories in a path
pathsep	String to separate paths in a path set string
linesep	String to separate text lines
environ	The current environment string

The sys module has some important functions:

Function	**Comment(s)**
getcwd()	Get the current directory
mkdir(path) makedirs(path) rmdir(path)	Create/delete a directory
remove(path) -- or -- unlink(path)	Delete a file
listdir(path)	List the files in a path
rename(path, new)	Renames a file/directory to new
system(command)	Run a shell command

See the *Python Library Reference* for more information.

Appendix G: The os.path module

The os.path module has some important functions:

Function	**Comment(s)**
exists(path)	True is path exists
abspath(path) normpath(path) normcase(path)	Returns the absolute form of the path Returns the normalized form of the path Returns the path in the normal case
basename(path) dirname(path)	Returns the file part of path Returns the directory part of path
commonprefix(list)	Returns the longest common prefix of the paths in the list
gethome()	Gets the home directory
getsize(path)	Gets the size of the path file
isabs(path) isfile(path) isdir(path)	Tests to see if path is absolute Tests to see if path is a file Tests to see if path is a dir
samepath(path1, path2)	True if path1 and path2 represent the same file
join(list)	Joins the path elements in the list
split(path) splitdrive(path) splitext(path)	Returns (path, last_element) Returns (drive, rest_of_path) Returns (root, extension)

See the *Python Library Reference* for more information.

Appendix H: Regular expression control characters

The most useful Regular Expression special characters are:

Special Notation	**Comment(s)**
Any character except those	Matches that character

<table>
<tr><td>below</td><td></td></tr>
<tr><td>.</td><td>Matches any character</td></tr>
<tr><td>^</td><td>Matches the start of the string</td></tr>
<tr><td>$</td><td>Matches the end of the string</td></tr>
<tr><td>?
??</td><td>Matches longest 0 or 1 of the proceeding
Matches shortest 0 or 1 of the proceeding</td></tr>
<tr><td>+
+?</td><td>Matches longest 1 or more of the proceeding
Matches shortest 1 or more of the proceeding</td></tr>
<tr><td>*
*?</td><td>Matches longest 0 or more of the proceeding
Matches shortest 0 or more of the proceeding</td></tr>
<tr><td>{m,n}
{m,n}?</td><td>Matches longest m to n of the proceeding
Matches shortest m to n of the proceeding</td></tr>
<tr><td>[...]
[^...]</td><td>Defines the set of enclosed characters
Defines the set of non-enclosed characters</td></tr>
<tr><td>...|...</td><td>Matches a choice (or)</td></tr>
<tr><td>(...)
(?...)</td><td>Matches the sequence (or group) ...; groups are ordered from left to right with origin 1
Matches a sequence but does not define a group</td></tr>
<tr><td>(?P<name>...)
(?P=name)</td><td>Matches a sequence (or group) ... giving it a name
Matches the sequence defined with the name</td></tr>
<tr><td>(?=...)
(?!...)</td><td>Matches ... but does not consume the test
Matches not ... but does not consume the test</td></tr>
<tr><td>\c</td><td>Special characters:
\c literal escapes: .?*+&^$|()[]
\c function escapes: see below</td></tr>
</table>

See the *Python Library Reference* for more information.

Function escapes:

Function Escapes	**Comment(s)**
\A	Matches at start of line

\Z	Matches at end of line
\B \b	Matches not at beginning or end of a word Matches at beginning or end of a word
\D \d	Matches not any decimal digit (0..9) Matches any decimal digit (0..9)
\S \s	Matches not any white space Matches any white space
\W \w	Matches not any alpha-numeric characters Matches any alpha-numeric characters
\#	Matches group #

Several options exist to modify how regular expression are processed. Options are bit flags and may be combined by OR-ing (|) them together. Some of the more useful options are:

Option	**Comment(s)**
IGNORECASE -- or -- I	Match ignoring case
MULTILINE -- or -- M	Causes '^' and '$' to match internal line boundaries (vs. just the start and end of the string)
DOTALL -- or -- S	Causes '.' to match a newline

Appendix I: Generated factor.java

The following is the code generated by `jythonc` compiler for the *factor.py* file of The factorial engine: factor.py on page 67 .

```
import org.python.core.*;
```

```
public class factor extends java.lang.Object {
     static String[] jpy$mainProperties =
         new String[] {"python.modules.builtin",
                       "exceptions:org.python.core.exceptions"};

     static String[] jpy$proxyProperties =
         new String[] {"python.modules.builtin",
                       "exceptions:org.python.core.exceptions",
                       "python.options.showJavaExceptions",
                       "true"};

     static String[] jpy$packages = new String[] {};

     public static class _PyInner extends PyFunctionTable
            implements PyRunnable {
         private static PyObject i$0;
         private static PyObject i$1;
         private static PyObject s$2;
         private static PyObject l$3;
         private static PyObject i$4;
         private static PyObject s$5;
         private static PyObject s$6;
         private static PyObject s$7;
         private static PyObject s$8;
         private static PyObject s$9;
         private static PyObject i$10;
         private static PyObject i$11;
         private static PyObject s$12;
         private static PyFunctionTable funcTable;
         private static PyCode c$0___init__;
         private static PyCode c$1_addListener;
         private static PyCode c$2_addListeners;
         private static PyCode c$3_removeListener;
         private static PyCode c$4_removeListeners;
         private static PyCode c$5_fireListeners;
         private static PyCode c$6_cancel;
         private static PyCode c$7_calculate;
         private static PyCode c$8_Factorial;
         private static PyCode c$9_doFac;
         private static PyCode c$10_main;
         private static void initConstants() {
             i$0 = Py.newInteger(0);
             i$1 = Py.newInteger(1);
             s$2 = Py.newString("only positive integers supported: ");
             l$3 = Py.newLong("1");
             i$4 = Py.newInteger(100);
             s$5 = Py.newString("__main__");
             s$6 = Py.newString("running...");
             s$7 = Py.newString("For");
             s$8 = Py.newString("result =");
             s$9 = Py.newString("Exception -");
             i$10 = Py.newInteger(10);
             i$11 = Py.newInteger(1000);
             s$12 = Py.newString("C:\\Articles\\factor.py");
             funcTable = new _PyInner();
             c$0___init__ = Py.newCode(1, new String[] {"self"},
                                       "C:\\Articles\\factor.py",
                                       "__init__", false, false,
```

```
                          funcTable, 0,
                          null, null, 0, 1);
c$1_addListener = Py.newCode(2,
                             new String[]
                             {"self", "listener", "ll"},
                             "C:\\Articles\\factor.py",
                             "addListener", false,
                             false, funcTable, 1,
                             null, null, 0, 1);
c$2_addListeners = Py.newCode(2,
                             new String[]
                             {"self", "listeners", "l"},
                             "C:\\Articles\\factor.py",
                             "addListeners", false,
                             false, funcTable, 2,
                             null, null, 0, 1);
c$3_removeListener = Py.newCode(2,
                             new String[]
                             {"self", "listener", "ll"},
                             "C:\\Articles\\factor.py",
                             "removeListener", false,
                             false, funcTable, 3,
                             null, null, 0, 1);
c$4_removeListeners = Py.newCode(2,
                             new String[]
                             {"self", "listeners", "l"},
                             "C:\\Articles\\factor.py",
                             "removeListeners", false,
                             false, funcTable, 4,
                              null, null, 0, 1);
c$5_fireListeners = Py.newCode(2,
                             new String[]
                             {"self", "value", "func"},
                             "C:\\Articles\\factor.py",
                             "fireListeners", false,
                             false, funcTable, 5,
                             null, null, 0, 1);
c$6_cancel = Py.newCode(1,
                             new String[]
                             {"self"},
                             "C:\\Articles\\factor.py",
                             "cancel", false,
                             false, funcTable, 6,
                             null, null, 0, 1);
c$7_calculate = Py.newCode(2,
                             new String[]
                             {"self", "value", "next",
                              "x", "last", "result"},
                             "C:\\Articles\\factor.py",
                             "calculate", false,
                             false, funcTable, 7,
                             null, null, 0, 1);
c$8_Factorial = Py.newCode(0,
                             new String[]
                             {},
                             "C:\\Articles\\factor.py",
                             "Factorial", false,
                             false, funcTable, 8,
```

```
                                        null, null, 0, 0);
        c$9_doFac = Py.newCode(1,
                                        new String[]
                                        {"value", "e"},
                                        "C:\\Articles\\factor.py",
                                        "doFac", false,
                                        false, funcTable, 9,
                                        null, null, 0, 1);
        c$10_main = Py.newCode(0,
                                        new String[] {},
                                        "C:\\Articles\\factor.py",
                                        "main", false,
                                        false, funcTable, 10,
                                        null, null, 0, 0);
    }


    public PyCode getMain() {
        if (c$10_main == null) _PyInner.initConstants();
        return c$10_main;
    }

    public PyObject call_function(int index, PyFrame frame) {
        switch (index){
            case 0:
            return _PyInner.__init__$1(frame);
            case 1:
            return _PyInner.addListener$2(frame);
            case 2:
            return _PyInner.addListeners$3(frame);
            case 3:
            return _PyInner.removeListener$4(frame);
            case 4:
            return _PyInner.removeListeners$5(frame);
            case 5:
            return _PyInner.fireListeners$6(frame);
            case 6:
            return _PyInner.cancel$7(frame);
            case 7:
            return _PyInner.calculate$8(frame);
            case 8:
            return _PyInner.Factorial$9(frame);
            case 9:
            return _PyInner.doFac$10(frame);
            case 10:
            return _PyInner.main$11(frame);
            default:
            return null;
        }
    }

    private static PyObject __init__$1(PyFrame frame) {
        frame.getlocal(0).__setattr__("_Factorial__listeners",
                       new PyList(new PyObject[] {}));
        frame.getlocal(0).__setattr__("_Factorial__cancelled", i$0);
        return Py.None;
    }
```

```
    private static PyObject addListener$2(PyFrame frame) {
        frame.setlocal(2,
             frame.getlocal(0).__getattr__("_Factorial__listeners"));
        if (frame.getlocal(1)._notin(
               frame.getlocal(2) ).__nonzero__()) {
            frame.getlocal(2).invoke("append", frame.getlocal(1));
        }
        return Py.None;
    }

    private static PyObject addListeners$3(PyFrame frame) {
        // Temporary Variables
        int t$0$int;
        PyObject t$0$PyObject, t$1$PyObject;

        // Code
        t$0$int = 0;
        t$1$PyObject = frame.getlocal(1);
        while ((t$0$PyObject =
               t$1$PyObject.__finditem__(t$0$int++)) != null) {
            frame.setlocal(2, t$0$PyObject);
            frame.getlocal(0).invoke("addListener",
                frame.getlocal(2));
        }
        return Py.None;
    }

    private static PyObject removeListener$4(PyFrame frame) {
        frame.setlocal(2,
frame.getlocal(0).__getattr__("_Factorial__listeners"));
        frame.getlocal(2).invoke("remove", frame.getlocal(1));
        return Py.None;
    }

    private static PyObject removeListeners$5(PyFrame frame) {
        // Temporary Variables
        int t$0$int;
        PyObject t$0$PyObject, t$1$PyObject;

        // Code
        t$0$int = 0;
        t$1$PyObject = frame.getlocal(1);
        while ((t$0$PyObject =
 t$1$PyObject.__finditem__(t$0$int++)) != null) {
            frame.setlocal(2, t$0$PyObject);
            frame.getlocal(0).invoke("removeListener",
                   frame.getlocal(2));
        }
        return Py.None;
    }

    private static PyObject fireListeners$6(PyFrame frame) {
        // Temporary Variables
        int t$0$int;
        PyObject t$0$PyObject, t$1$PyObject;

        // Code
        t$0$int = 0;
```

```
            t$1$PyObject =
               frame.getlocal(0).__getattr__("_Factorial__listeners");
            while ((t$0$PyObject =
                   t$1$PyObject.__finditem__(t$0$int++)) != null) {
                frame.setlocal(2, t$0$PyObject);
                frame.getlocal(2).__call__(frame.getlocal(1));
            }
            return Py.None;
        }

        private static PyObject cancel$7(PyFrame frame) {
            frame.getlocal(0).__setattr__("_Factorial__cancelled", i$1);
            return Py.None;
        }

        private static PyObject calculate$8(PyFrame frame) {
            // Temporary Variables
            int t$0$int;
            PyObject t$0$PyObject, t$1$PyObject;

            // Code
            if (((t$0$PyObject = frame.getglobal("type").
                 __call__(frame.getlocal(1)).
                 _ne(frame.getglobal("types").
                 __getattr__("IntType"))).__nonzero__()
               ? t$0$PyObject
               : frame.getlocal(1)._lt(i$0)).__nonzero__()) {
                throw Py.makeException(
                   frame.getglobal("ValueError"),
                   s$2._add(frame.getglobal("str").
                      __call__(frame.getlocal(1))));
            }
            frame.getlocal(0).__setattr__("_Factorial__cancelled", i$0);
            frame.setlocal(5, l$3);
            frame.getlocal(0).invoke("fireListeners", i$0);
            if (frame.getlocal(1)._le(i$1).__nonzero__()) {
                frame.setlocal(5, l$3);
            }
            else {
                frame.setlocal(4, i$0);
                t$0$int = 0;
                t$1$PyObject = frame.getglobal("range").
                  __call__(i$1,frame.getlocal(1)._add(i$1));
                while ((t$0$PyObject = t$1$PyObject.
                          __finditem__(t$0$int++)) != null) {
                    frame.setlocal(3, t$0$PyObject);
                    if (frame.getlocal(0).
                        __getattr__("_Factorial__cancelled").__nonzero__()) {
                        break;
                    }
                    frame.setlocal(5,
                       frame.getlocal(5)._mul(frame.getlocal(3)));
                    frame.setlocal(2,
                       frame.getlocal(3)._mul(i$4)._div(frame.getlocal(1)));
                    if
(frame.getlocal(2)._ne(frame.getlocal(4)).__nonzero__()) {
                        frame.getlocal(0).invoke("fireListeners",
                           frame.getlocal(2));
```

```
                    frame.setlocal(4, frame.getlocal(2));
                }
            }
        }
        frame.getlocal(0).invoke("fireListeners", i$4);
        if (frame.getlocal(0).
            __getattr__("_Factorial__cancelled").__nonzero__()) {
            frame.setlocal(5, i$1.__neg__());
        }
        return frame.getlocal(5);
    }

    private static PyObject Factorial$9(PyFrame frame) {
        frame.setlocal("__init__",
            new PyFunction(frame.f_globals,
                           new PyObject[] {}, c$0___init__));
        frame.setlocal("addListener",
            new PyFunction(frame.f_globals,
                           new PyObject[] {}, c$1_addListener));
        frame.setlocal("addListeners",
            new PyFunction(frame.f_globals,
                           new PyObject[] {}, c$2_addListeners));
        frame.setlocal("removeListener",
            new PyFunction(frame.f_globals,
                           new PyObject[] {}, c$3_removeListener));
        frame.setlocal("removeListeners",
            new PyFunction(frame.f_globals,
                           new PyObject[] {}, c$4_removeListeners));
        frame.setlocal("fireListeners",
           new PyFunction(frame.f_globals,
                          new PyObject[] {}, c$5_fireListeners));
        frame.setlocal("cancel",
           new PyFunction(frame.f_globals,
                          new PyObject[] {}, c$6_cancel));
        frame.setlocal("calculate",
           new PyFunction(frame.f_globals,
                          new PyObject[] {}, c$7_calculate));
        return frame.getf_locals();
    }

    private static PyObject doFac$10(PyFrame frame) {
        // Temporary Variables
        PyException t$0$PyException;

        // Code
        try {
            Py.printComma(s$7);
            Py.printComma(frame.getlocal(0));
            Py.printComma(s$8);
            Py.println(frame.getglobal("fac").
               invoke("calculate", frame.getlocal(0)));
        }
        catch (Throwable x$0) {
            t$0$PyException = Py.setException(x$0, frame);
            if (Py.matchException(t$0$PyException,
                                   frame.getglobal("ValueError"))) {
                frame.setlocal(1, t$0$PyException.value);
                Py.printComma(s$9);
```

```
                    Py.println(frame.getlocal(1));
                }
                else throw t$0$PyException;
            }
            return Py.None;
        }

        private static PyObject main$11(PyFrame frame) {
            frame.setglobal("__file__", s$12);

            frame.setlocal("sys",
                           org.python.core.imp.importOne("sys", frame));
            frame.setlocal("types",
                           org.python.core.imp.importOne("types", frame));
            frame.setlocal("exceptions",
                           org.python.core.imp.importOne("exceptions", frame));
            frame.setlocal("Factorial",
                           Py.makeClass("Factorial",
                                        new PyObject[] {},
c$8_Factorial, null));
            if (frame.getname("__name__")._eq(s$5).__nonzero__()) {
                Py.printComma(frame.getname("sys").
                              __getattr__("argv").__getitem__(i$0));
                Py.println(s$6);
                frame.setlocal("fac",
                               frame.getname("Factorial").__call__());
                frame.setlocal("doFac",
                               new PyFunction(frame.f_globals,
                                              new PyObject[] {}, c$9_doFac));
                frame.getname("doFac").__call__(i$1.__neg__());
                frame.getname("doFac").__call__(i$0);
                frame.getname("doFac").__call__(i$1);
                frame.getname("doFac").__call__(i$10);
                frame.getname("doFac").__call__(i$4);
                frame.getname("doFac").__call__(i$11);
            }
            return Py.None;
        }

    }
    public static void moduleDictInit(PyObject dict) {
        dict.__setitem__("__name__", new PyString("factor"));
        Py.runCode(new _PyInner().getMain(), dict, dict);
    }

    public static void main(String[] args) throws java.lang.Exception {
        String[] newargs = new String[args.length+1];
        newargs[0] = "factor";
        System.arraycopy(args, 0, newargs, 1, args.length);
        Py.runMain(factor._PyInner.class, newargs,
                   factor.jpy$packages,
                   factor.jpy$mainProperties, null,
                   new String[] {"factor"});
    }

}
```

Note: The above code has been reformatted for line length.

Appendix J: Formatting strings and values

Note that a simplified form of Appendix J originally appeared as multiple panels in Part 1 of this tutorial.

Jython strings support a special formating operation similar to C's `printf`, but using the modulo ("%") operator. The right-hand set of items is substituted into the left-hand string at the matching `%x` locations in the string. The set value is usually a single value, a tuple of values, or a dictionary of values.

The general format of the format specification is

```
%{(key)}{flag}...{width}{.precision}x
```

Here's a guide to the format items:

- **key**: Optional key to lookup in a supplied dictionary
- **flag**: Optional flags (reasonable combinations supported)
  - **#**: Display any special format prefix (for example, "0" for octal, "0x" for hex)
  - **+**: Display a "+" on positive numbers
  - **blank**: Display a leading space on positive numbers
  - **-**: Left (vs. right) justify the value in the width
  - **0**: Left pad with "0" (vs. spaces)
- **width**: Minimum width of the field (will be longer for large values)
- **precision**: Number of digits after any decimal point
- **x**: Format code as described below

The format operator supports the following format characters:

Character(s)	Result Format	Comment(s)

%s, %r	String	%s does `str(x)`, %r does `repr(x)`
%i, %d	Integer Decimal	Basically the same format
%o, %u, %x, %X	Unsigned Value	In octal, unsigned decimal, hexadecimal
%f, %F	Floating Decimal	Shows fraction after decimal point
%e, %E, %g, %G	Exponential	%g is %f unless the value is small; else %e
%c	Character	Must be a single character or integer
%%	Character	The % character

Note: more details on the structure and options of the format item can be found in the Python Library Reference (Resources on page 73). Use of case in format characters (for example, *X* vs *x* causes the symbol to show in matching case.

For example

```
print "%s is %i %s %s than %s!" % ("John", 5, "years", "older", "Mark")

print "Name: %(last)s, %(first)s" % \
        {'first':"Barry", 'last':"Feigenbaum", 'age':18}
```

prints

```
John is 5 years older than Mark!
Name: Feigenbaum, Barry
```

Appendix K: Built-in functions

Note that Appendix K appeared in Part 1 of this tutorial.

Jython provides very useful built-in functions that can be used without any imports. The most commonly used ones are summarized below:

Syntax	**Use/Comment(s)**	**Example(s)**
abs(x)	Absolute value	abs(-1) --> 1
apply(func, pargs {, kargs}) -- or --	Execute the function with the supplied positional arguments and optional keyword arguments	apply(lambda x, y: x * y, (10, 20)) --> 200

func(*pargs {, **kargs})		
callable(x)	Tests to see if the object is callable (i.e, is a function, class or implements __call__)	callable(MyClass) --> 1
chr(x)	Converts the integer (0 - 65535) to a 1-character string	chr(9) --> "\t"
cmp(x, y)	Compares x to y: returns: negative if x < y; 0 if x == y; positive if x > y	cmp("Hello", "Goodbye") --> > 0
coerce(x, y)	Returns the tuple of x and y coerced to a common type	coerce(-1, 10.2) --> (-1.0, 10.2)
compile(text, name, kind)	Compile the text string from the source name. Kind is: "exec", "eval" or "single"	x = 2 c = compile("x * 2", "<string>", "eval") eval(c) --> 4
complex(r, i)	Create a complex number	complex(1, 2) --> 1.0+2.0j complex("1.0-0.1j") --> 1.0-0.1j
dir({namespace})	Returns a list of the keys in a namespace (local if omitted)	dir() --> [n1, ..., nN]
vars({namespace})	Returns the namespace (local if omitted); do not change it	vars() --> {n1:v1, ..., nN:vN}
divmod(x, y)	Returns the tuple (x /y, x % y)	divmod(100, 33) --> (3, 1)
eval(expr {, globals {, locals}})	Evaluate the expression in the supplied namespaces	myvalues = {'x':1, 'y':2} eval("x + y", myvalues) --> 3
execfile(name {,globals {, locals}})	Read and execute the named file in the supplied namespaces	execfile("myfile.py")
filter(func, list)	Creates a list of items for which func returns true	filter(lambda x: x > 0, [-1, 0, 1, -5, 10]) --> [1, 10]
float(x)	Converts x to a float	float(10) --> 10.0 float("10.3") --> 10.3
getattr(object, name {, default})	Gets the value of the object's attribute; if not defined return default (or an exception if no default)	getattr(myObj, "size", 0) --> 0
setattr(object,	Creates/sets the value of the	setattr(myObj, "size", 10)

name, value)	object's attribute	
hasattr(object, name)	Test to see if the object has an attribute	hasattr(myObj, "size") --> 0
globals()	Returns the current global namespace dictionary	{n1:v1, ..., nN:vN}
locals()	Returns the current local namespace dictionary	{n1:v1, ..., nN:vN}
hash(object)	Returns the object's hash value. Similar to `java.lang.Object.hashCode()`	hash(x) --> 10030939
hex(x)	Returns a hex string of x	hex(-2) --> "FFFFFFFE"
id(object)	Returns a unique stable integer id for the object	id(myObj) --> 39839888
input(prompt)	Prompts and evaluates the supplied input expression; equivalent to `eval(raw_input(prompt))`	input("Enter expression:") with "1 + 2" --> 3
raw_input(prompt)	Prompts for and inputs a string	raw_input("Enter value:") with "1 + 2" --> "1 + 2"
int(x{, radix})	Converts to an integer; radix: 0, 2..36; 0 implies guess	int(10.2) --> 10 int("10") --> 10 int("1ff", 16) --> 511
isinstance(object, class)	Tests to see if object is an instance of class or a subclass of class; class may be a tuple of classes to test multiple types	isinstance(myObj, MyObject) --> 0 isinstance(x, (Class1, Class2)) --> 1
issubclass(xclass, clsss)	Tests to see if xclass is a sub-(or same) class of class; class may be a tuple of classes to test multiple types	issubclass(MyObject, (Class1, Class2)) --> 0
len(x)	Returns the length (number of items) in the sequence or map	len("Hello") --> 5
list(seq)	Converts the sequence into a list	list((1, 2, 3)) --> [1,2,3] list("Hello") --> ['H','e','l','l','o']
tuple(seq)	Converts the sequence into a tuple	tuple((1, 2, 3)) --> (1,2,3) tuple("Hello")--> ('H','e','l','l','o')
long(x {, radix})	Converts to a long integer; radix: 0, 2..36; 0 implies guess	long(10) --> 10L

		long("10000000000") --> 10000000000L
map(func, list, ...)	Creates a new list from the results of applying func to each element of each list	map(lambda x,y: x+y, [1,2],[3,4]) --> [4,6] map(None, [1,2],[3,4]) --> [[1,3],[2,4]]
max(x)	Returns the maximum value	max(1,2,3) --> 3 max([1,2,3]) --> 3
min(x)	Returns the minimum value	min(1,2,3) --> 1 min([1,2,3]) --> 1
oct(x)	Converts to an octal string	oct(10) --> "012 oct(-1) --> "037777777777"
open(name, mode {, bufsize})	Returns an open file. Mode is:(r|w|a){+}{b}	open("useful.dat", "wb", 2048)
ord(x)	Returns the integer value of the character	ord('\t') --> 9
pow(x,y) pow(x,y,z)	Computes x ** y Computes x ** y % z	pow(2,3) --> 8
range({start,} stop {, inc}) xrange({start,} stop {, inc})	Returns a sequence ranging from start to stop in steps of inc; start defaults to 0; inc defaults to 1. Use xrange for large sequences (say more than 20 items)	range(10) --> [0,1,2,3,4,5,6,7,8,9] range(9,-1,-1) --> [9,8,7,6,5,4,3,2,1,0]
reduce(func, list {, init})	Applies func to each pair of items in turn accumulating a result	reduce(lambda x,y:x+y, [1,2,3,4],5) --> 15 reduce(lambda x,y:x&y, [1,0,1]) --> 0 reduce(None, [], 1) --> 1
repr(object) -- or -- \`object\`	Convert to a string from which it can be recreated, if possible	repr(10 * 2) --> "'20'" repr('xxx') --> "'xxx'" x = 10; \`x\` --> "10'"
round(x {, digits})	Rounds the number	round(10.009, 2) --> 10.01 round(1.5) --> 2
str(object)	Converts to human-friendly string	str(10 * 2) --> "20" str('xxx') --> 'xxx'

type(object)	Returns the type (not the same as class) of the object. To get the class use `object.__class__`. Module *types* has symbolic names for all Jython types	x = "1"; type(x) is type('') --> 1
zip(seq, ...)	Zips sequences together; results is only as long as the shortest input sequence	zip([1,2,3],"abc") --> [(1,'a'),(2,'b'),(3,'c')]

Appendix L: Jython types summary

Note that Appendix L appeared in Part 1 of this tutorial.

Jython supports many object types. The module *types* defines symbols for these types. The function *type* gets the type of any object. The type value can be tested (see on page). The table below summarizes the most often used types.

Type symbol	**Jython runtime type**	**Comment(s)**
ArrayType	PyArray	Any array object
BuiltinFunctionType	PyReflectedFunction	Any built-in function object
BuiltinMethodType	PyMethod	Any built-in method object
ClassType	PyClass	Any Jython class object
ComplexType	PyComplex	Any complex object
DictType -- or -- DictionaryType	PyDictionary	Any dictionary object
FileType	PyFile	Any file object
FloatType	PyFloat	Any float object
FunctionType	PyFunction	Any function object
InstanceType	PyInstance	Any class instance object
-- none --	PyJavaInstance	Any Java class instance object
IntType	PyInteger	Any integer object
LambdaType	PyFunction	Any lambda function

		expression object
ListType	PyList	Any list object
LongType	PyLong	Any long object
MethodType	PyMethod	Any non-built-in method object
ModuleType	PyModule	Any module object
NoneType	PyNone	Any `None` (only one) object
StringType	PyString	Any ASCII string object
TracebackType	PyTraceback	Any exception traceback object
TupleType	PyTuple	Any tuple object
TypeType	PyJavaClass	Any *type* object
UnboundMethodType	PyMethod	Any method (without a bound instancee) object
UnicodeType	PyString	Any Unicode string object
XRangeType	PyXRange	Any extended range object

Note: several types map to the same Java runtime type. For more information on types see the Python Library Reference (Resources on page 73).

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.