Intro to Jython, Part 1: Java programming made easier

Barry Feigenbaum
Sr. Consulting IT Architect
IBM

08 April 2004

This is the first in a two-part tutorial that will introduce you to the Jython scripting language and provide you with enough knowledge to begin developing your own Jython-based applications. In this first half of the tutorial, you'll learn the concepts and programming basics of working with Jython, including access options and file compilation, syntax and data types, program structure, procedural statements, and functions.

About this tutorial

What is this tutorial about?

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Explore alternative languages for the Java platform

This two-part tutorial will introduce you to the Jython scripting language, and provide you with enough knowledge to begin developing your own Jython-based applications. 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.

Jython is extremely useful because it provides the productivity features of a mature scripting language while running on a JVM. Unlike a Python program, a Jython program can run in *any* environment that supports a JVM. Today, this means most major computing systems, including Microsoft Windows, Mac OS, most UNIX variants including all Linux systems, and all IBM systems.

This tutorial covers Jython in progressive detail. In this first half of the tutorial, we'll cover the concepts and programming basics of working with Jython, including access options and file compilation, syntax and data types, program structure, procedural statements, and functions. The second half of the tutorial will start with a conceptual introduction to object-oriented programming in Jython. From there, we'll move on to a more hands-on discussion, encompassing class statements, attributes, and methods, abstract classes, and operator overloading. This advanced discussion will also include debugging, string processing, file I/O, and Java support in Jython.

The tutorial will conclude with a step-by-step demonstration of how to build a working GUI app in Jython.

The example code will be very simple in the beginning of the tutorial, but by the end of the second half you will be up and running with complete functions, classes, and programs. Included with the tutorial is a set of appendices detailing the inner workings of Jython.

Should I take this tutorial?

This tutorial is oriented towards software developers at all levels, from casual dabblers to professionals. It is especially oriented towards Java developers who want to leverage the productivity advantages of a scripting language. It is also targeted towards Visual Basic and C++/C# programmers who want an easier entry into the Java development world.

Together, we will cover the following aspects of scripting with Jython:

Part 1

- Download and installation
- A conceptual introduction to Jython
- Working from the command-line vs source files
- Syntax and data types
- Program structure
- Procedural statements
- Functions

Part 2

- Object-oriented programming with Jython
- Debugging
- Java support
- String processing
- File IO
- Building a 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. It will also be helpful to have a working knowledge of the Java platform, as 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 Resources to download the latest version of the Java development kit (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.

Getting started

Installation instructions

In this section we'll walk through each of the steps for downloading, installing, and verifying Jython on your development system.

Download Jython

You can download Jython 2.1 from the Jython home page. You'll find easy-to-follow download instructions on the download page.

As previously mentioned, this tutorial is based on the current stable Jython level, which is version 2.1. More advanced development levels may also be available from the Jython home page.

Install Jython

Installing Jython is simple: just execute the class file you've downloaded from the Jython homepage. Assuming that you have a JRE installed and have the downloaded class file in your current directory (C:\ in the examples below) the following command will install Jython (note that *<java_home>* is the directory the JRE is installed in):

```
C:\><java_home>\bin\java jython-21
```

Please follow the install application's prompts. I recommend you select the defaults, and that you select *c:\Jython-2.1* as the destination directory.

Verify the install

To verify that Jython is installed, enter the command:

```
C:\>dir c:\Jython-2.1
```

The result should be a listing like this one:

```
  Volume in drive C is C_DRIVE
  Volume Serial Number is ????-????

  Directory of C:\Jython-2.1

--/--/----  --:---      <DIR>          .
--/--/----  --:---      <DIR>          ..
--/--/----  --:---                1,873 ACKNOWLEDGMENTS
--/--/----  --:---      <DIR>          cachedir
--/--/----  --:---      <DIR>          com
--/--/----  --:---      <DIR>          Demo
--/--/----  --:---      <DIR>          Doc
--/--/----  --:---      <DIR>          installer
--/--/----  --:---                  428 jython.bat
--/--/----  --:---              719,950 jython.jar
--/--/----  --:---                  272 jythonc.bat
--/--/----  --:---      <DIR>          Lib
--/--/----  --:---                7,184 LICENSE.txt
--/--/----  --:---               18,178 NEWS
--/--/----  --:---      <DIR>          org
--/--/----  --:---                  651 README.txt
--/--/----  --:---                4,958 registry
--/--/----  --:---      <DIR>          Tools
--/--/----  --:---              224,493 Uninstall.class
               9 File(s)        977,987 bytes
              ? Dir(s)   ??? bytes free
```

A test run

The final step is to ensure that Jython is configured. To run Jython, start by entering the command:

```
C:\>c:\jython-2.1\jython
```

The command should result in an introduction similar to this one:

```
Jython 2.1 on java1.4.1_01 (JIT: null)
Type "copyright", "credits" or "license" for more information.
```

Finally, we'll exit Jython. At the Jython prompt, enter the following command:

```
>>> import sys; sys.exit()
```

Alternatively, you could just press **Ctrl+C** two times.

Making life more convenient

There is just one last thing you should know before we close this section on getting started. You can eliminate the need to specify the Jython command path (`<d>:\jython-2.1`) by adding it to your `PATH` variable. Now you can just type **jython** at the command prompt.

Basic concepts and advantages of Jython

What is Jython?

As previously mentioned, Jython is an implementation of Python written in the Java language and integrated with the Java platform. Python is a scripting language often used in UNIX-based

systems, including Linux. Python was invented by Guido van Rossum and introduced to the developer community in 1991. Jython currently supports the Python syntax at level 2.1.

What is a scripting language?

Unlike the Java programming language, Jython is a scripting language. A scripting language is generally defined as follows:

- Very easy to learn and code
- Expressive and concise, yet powerful
- Has minimal required structure to create a running "program"
- Supports interactive (command-at-a-time) execution
- Does not require a compile step
- Supports reflective programming
- Supports functional programming
- Supports dynamic execution of source (that is, an *eval* function)
- Runs external programs

In general, it can be said that scripting languages value programmer efficiency over machine efficiency and performance. Compared to a programming language such as the Java language, Jython is easy to learn and efficient to code.

Jython can also be described as an agile language. Agile languages are generally thought of as being capable of performing a wide variety of tasks and useful for many different types of problems, easy-to-use and yet powerful and expressive. They are also ideal rapid prototyping languages.

Advantages of Jython

Like its C-based cousin Python, Jython is most at home when used to develop small programs and scripts; it has many features that allow simple but functional programs to be created in a few minutes. This does not mean Jython cannot be used for large-scale programming. In fact, Jython supports a sophisticated packaging scheme, similar to that of the Java language. By virtue of its object-oriented nature, Jython is highly extendable and provides the latest constructs for effective software engineering.

Like the Java language and unlike some other scripting languages such as Perl and Rexx, Jython was designed to be an object-oriented language from the start. Thus, it offers powerful object-oriented programming (OOP) features that are easy to understand and use.

One of Jython's biggest advantages is that it runs on any JVM, so applications coded in Jython can run on almost any computing system.

Jython and the Java platform

Jython is built on the Java platform. From the platform's point of view, the Jython runtime is just another Java class. This is quite apparent if you look into the JYTHON.BAT file, which launches the Java runtime with the Jython interpreter as its main class, as shown below:

```
@echo off
rem This file generated by Jython installer
rem
JAVA_HOME=<java_home>
rem
rem collect all arguments into %ARGS%
set ARGS=
:loop
if [%1] == [] goto end
     set ARGS=%ARGS% %1
     shift
     goto loop
:end

%JAVA_HOME%\bin\java.exe
    -Dpython.home=C:\jython-2.1
    -cp "C:\jython-2.1\jython.jar;%CLASSPATH%"
    org.python.util.jython %ARGS%
```

Everything is interpreted

At its heart Jython is an *interpreted language*. In Jython, there is no pre-compile step as there is in Java and C++; each time Jython code is run it is interpreted afresh. As such, code changes can be very quickly made and tested. Jython code can also be entered interactively (that is, one line at a time). Furthermore, you can dynamically construct Jython code (that is, as a string) and execute it directly. This enables coding flexibility not possible in Java coding.

The Jython interpreter converts Jython source into an internal form for more efficient processing. It does this during a first pass that verifies syntax. Once this pass is complete the internalized source is interpreted. Jython also caches this internalized form on disk. In a Java class file for the Jython module `<name>.py`, the cached file would be `<name>$py.class`.

Interpretation does have its disadvantages, although most are minor. For example, use of an undefined variable is not a compiler error, so it will be detected only if (and when) the statement in which the variable is used is executed. While this can seem a disadvantage when compared to compiled languages, the fact that you can edit and then immediately run a program and experience the error (if it exists) makes up for it. A simple test-and-debug procedure takes about as much time as repeated edit-compile steps do to remove an error.

About performance

Because Jython is interpreted, it can be slower than a compiled language such as Java. In most applications, such as scripts or GUIs, this difference is hardly noticeable. In most cases, Jython's increased design and coding flexibility more than makes up for any small performance loss.

Because Jython code is dynamically converted to Java byte code, the latest enhancements to the Java platform (such as JITs and Sun's HotSpot JVM) can also eliminate many performance issues.

For an additional performance boost it is possible to implement code sections in the Java language and call them from Jython. For example, you could prototype your programs in Jython, test them out, and (in the case of performance issues) convert the critical sections to Java code. This technique is a good combination of the powers of Jython and the Java language, as prototyping is

much easier in Jython than in Java. We'll talk more about combining the Java language and Jython in Part 2 of this tutorial.

Working with Jython

Using Jython as a command-line interpreter

One of the easiest ways to use Jython is as a command-line interpreter. In this manner, lines of code are entered one line at a time and you can see the results immediately. This is an ideal way to learn Jython and to try out new coding techniques with minimal overhead.

We'll start with a brief Jython interactive session. Enter the following commands after the ">>>" or "..." prompts:

```
C:\>c:\jython-2.1\jython
```

You should receive output that looks something like this:

```
Jython 2.1 on java1.4.1_01 (JIT: null)
Type "copyright", "credits" or "license" for more information.
>>> 1 + 2
3
>>> "Hello" + "Goodbye"
'HelloGoodbye'
>>> def fac(x):
...  if x <= 1: return 1
...  return long(x) * fac(x-1)
...
>>> fac(3)
6L
>>> fac(100)
933262154439441526816992388562667004907159682643816214685929638952175999
932299156089414639761565182862536979208272237582511852109168640000000
0000000000000000L
>>> import sys; sys.exit(0)
C:\>
```

With this example you can see how input is immediately executed. This includes simple expressions and more complex actions such as function definitions (that is, the `fac` function). Defined values and functions are available for immediate use. Notice, also, that Jython supports very large integers via the `long` type.

Note that in the above example the indentation of the `fac` function is critical. You'll learn more about this requirement later in the tutorial (see Blocks).

Using Jython via source files

If Jython accepted only command-line input it wouldn't be all that useful; thus, it also accepts source files. Jython source files end in the extension .py. A Jython file must contain a sequence of Jython statements. Expressions, such as *1 + 2*, are not valid statements (they execute but produce no displayed output).

To display expressions, you must place them in a `print` statement. Thus, the sequence from the previous section could be coded in a source file as follows:

```
print 1 + 2
print "Hello" + "Goodbye"
def fac(x):
  if x <= 1: return 1
  return long(x) * fac(x-1)
print fac(3)
print fac(100)
```

The above code would produce the same output as the examples in Using Jython as a command-line interpreter. In fact, the statements could have been entered interactively (with the addition of a blank line after the `fac` function) and would result in the same output.

The print statement

As shown in the previous section, we use the `print` statement to print expressions. The statement has the following forms:

```
print expression {, expression}... {,}

-- or --

print
```

The `print` statement above can also contain a list of expressions separated by commas. Each such expression is output with a space automatically added between them. So that `print "Hello", "Goodbye"` outputs `Hello Goodbye`.

If a `print` statement ends in comma, no new-line is output. The line `print` by itself outputs a new-line.

A "Hello World" example

In Jython, the quintessential example program -- Hello World -- is a single-line file (say, hello.py), as shown here:

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

To run the program you would enter the command: `C:\>c:\jython-2.1\jython hello.py`

Note that the .py extension is required; otherwise, a "file not found" error will occur. The `jython` command has several options. See the Jython home page (in Resources) for more information.

Jython source files are modules

Jython source files can contain more than a sequence of statements to execute. They can also contain function (see Jython functions) and class definitions (we'll talk more about class definitions in Part 2 of this tutorial). In fact, Jython source files can be *modules* (more on these later, in Modules and packages) that may not be used directly but instead *imported* by other programs. A single source file can perform both roles. Consider this variant of the file in the previous section:

```
def fac(x):
     if x <= 1: return 1
     return long(x) * fac(x-1)

if __name__ == "__main__":
     print 1 + 2
     print "Hello" + "Goodbye"
     print fac(3)
     print fac(100)
```

Again, running this file results in the same output as before. But if the file were imported into another program that only wanted to reuse the `fac` function, then none of the statements under the `if` (see The if statement) test would be executed.

Note also that each module has a name; the one directly executed from the command-line is called "`__main__`". This feature can be used to create a test case for each module.

Compiled Jython

Jython source files can be compiled to Java source code (which is automatically compiled into byte-code) to produce standalone class or Java Archive Files (JAR) files. This step is necessary to create Jython code that is called directly from the Java platform, such as when creating an applet or a servlet. It is also useful to provide Jython applications without releasing the Jython source.

Jython can be compiled into a pure Java class that can run directly on any JVM by use of the `jythonc` command (that is, assuming you have the necessary Jython JAR in the Java `CLASSPATH`). For more details on using `jythonc` see the Jython home page (Resources).

A compilation example

We'll use the *factor.py* file (see Download) as our example standalone program. To compile it, use the command:

```
c:\>c:\jython-2.1\jythonc factor.py
```

If there are no errors, Java class files `factor.class` and `factor$_PyInner.class` will be created. You'll find the actual generated Java source code in the download. To run this (now Java) application use the command:

```
c:\><java_home>\bin\java -cp .;c:\jython-2.1\jython.jar factor
```

The resulting output should look something like this:

```
factor 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 = 402387260077093773543702
     ... many digits removed ...
00000000000000000000
```

Note that the output is identical to that generated by using the `factor.py` program directly.

Jython basic data types

Everything is an object

Unlike the Java language, Jython sees everything, including all data and code, as an object. This means you can manipulate these objects using Jython code, making reflective and functional programming very easy to do in Jython. See Appendix G: Jython types summary for more information.

Some select types, such as numbers and strings, are more conveniently considered as values, not objects. Jython supports this notion as well.

Jython supports only one null value, with the reserved name of `None`.

Common operators

All Jython data types support the following fundamental operations:

Operation	Test usage	Comment(s)
`x and y`	Boolean and x with y	y is not evaluated if x is false Returns x or y as the result
`x or y`	Boolean or x with y	y is not evaluated if x is true Returns x or y as the result
`not x`	Boolean negation of x	Returns 0 or 1
`x < y`	Comparison strictly less than	Returns 0 or 1
`x > y`	Comparison strictly greater than	Returns 0 or 1
`x <= y`	Comparison less than or equal	Returns 0 or 1
`x >= y`	Comparison greater than or equal	Returns 0 or 1
`x == y`	Comparison equal	Returns 0 or 1
`x != y` `x <> y`	Comparison not equal	Returns 0 or 1
`x is y`	Sameness	Returns 1 if x is the same object as y; else 0
`x is not y`	Distinctness	Returns 1 if x is not the same object as y; else 0

Note that unlike in the Java language, all types are comparable. In general, if the types of the operands do not match, the result is unequal. The less-than or greater-than relations on complex types are consistent but arbitrary.

Boolean types

Jython has no separate boolean type. All the other types described in the following sections can be used as booleans. For numeric types, zero is considered to be false and all other values true. For structured types (that is, sequences and maps), an empty structure is considered to be false and others true. The `None` value is always false.

Numeric types

Numbers are *immutable* (that is, unchangeable after creation) objects treated as values. Jython supports three numeric types, as follows:

- **Integers** have no fractional part. Integers come in two subforms:
  - *Normal*: small values in the range -2**31 to 2**31 - 1 (like Java `int`s).
    Examples: -1, 0, 1, 10000
  - *Long*: large values limited only by the JVM's available memory (like Java `BigInteger`s).
    Examples: -1L, 0L, 1L, 10000000000000000000000000000000L
- **Floating point** values may have fractional parts. Floats support values identical to the Java `double` type.
  Examples: 0.0, -1.01, 2.5004E-100, -35e100
- **Complex values** are a pair of floating point values, called the *real* and *imaginary* part. If *x* is a complex value, then `x.real` is the real part and `x.imag` is the imaginary part. Either part may be 0.0. The method `x.conjugate` produces a new complex with `+x.real` and `-x.imag`.
  Examples: 1J, -1j, 1+2j, -3.7+2e5j

Additional numeric type operations and functions

Numeric types support the following additional operations and functions:

Operation/Function	Usage
`-x`	Negate x (that is, 0 - x)
`+x`	Posate - no change (that is, 0 + x)
`x + y`	Add y to x
`x - y`	Subtract y from x
`x * y`	Multiply x by y
`x / y`	Divide x by y
`x % y` `divmod(x, y)`	Take modulus of x by y Return (x / y, x % y)
`x ** y` `pow(x, y)`	Raise x to the y power Raise x to the y power
`abs(x)`	If x < 0, then -x; else x
`int(x)`	Convert x to an integer

`long(x)`	Convert x to a long
`float(x)`	Convert x to a float
`complex(r, i)` `complex(x)`	Convert r and i to a complex Convert x to a complex

Note: For numeric types, the operands are promoted to the next higher type. For integer operands /, %, and ** result in integer results. For the `int`, `long`, `float`, and `complex` conversion functions, *x* may be a string or any number.

Additional integer type operations

Integer types support the following additional operations:

Operation	Usage	Comment(s)
`x << y`	Shift x bits left by y	Similar to x * pow(2, y)
`x >> y`	Shift x bits right by y	Similar to x / pow(2, y)
`x & y`	And x and y bits	Clears the bits in x that are 0 in y.
`x | y`	Or x and y bits	Sets the bits in x that are 1 in y.
`x ^ y`	XOR x and y bits	Flips the bits in x that are 1 in y
`~x`	Invert x bits	Flips all bits

Additional floating type functions

Floating point types support the following additional functions (in module `math`):

Function	Comment(s)
`ceil(v)` `floor(v)`	Computes the ceiling and floor of v.

`sin(v)`
`cos(v)`
`tan(v)`
	Computes the sine, cosine, and tangent of v.

`acos(v)`
`asin(v)`
`atan(v)`
`atan2(v, w)`
	Computes the arcsine, arccosine, and arctangent of v (or v / w).

`sinh(v)`
`cosh(v)`
`tanh(v)`
	Computes the hyperbolic sine, cosine, and tangent of v.

`exp(v)`
`pow(v, w)`
`sqrt(v)`
`log(v)`
`log10(v)`
	Computes the powers and logarithms of v.

`fabs(v)`	Computes the absolute value of v.
`fmod(v, w)`	Computes the modulus of v and w. May not be the same as v % w.
`modf(v)`	Returns (as the tuple (i, f)) the integer and fractional parts of v (both as floats).
`frexp(v)`	Returns (as the tuple (m, e)) the float mantissa and integer exponent of v. The result is such that `v == m * 2 ** e`.

`ldexp(v, w)`	Computes `v * 2 ** w` (w must be an integer).
`hypot(v, w)`	Computes the hypotenuse of v and w (that is, `sqrt(v * v + w * w)`).

Math module examples

We'll run an example to demonstrate the functions in the `math` module from the previous section. See The import statement and Formatting strings and values for more information.

```
from math import *

print "PI = %f, e = %f" % (pi, e)

print "Sine of %f = %f" % (0.5, sin(0.5))
print "Cosine of %f = %f" % (0.5, cos(0.5))
print "Tangent of %f = %f" % (0.5, tan(0.5))
```

The example code results in the following output:

```
PI = 3.141593, e = 2.718282
Sine of 0.500000 = 0.479426
Cosine of 0.500000 = 0.877583
Tangent of 0.500000 = 0.546302
```

Jython collections

Collection types

Frequently, you will need to create collections of other data items. Jython supports two major collection types. The most basic is the *sequence* type which is an ordered collection of items. Sequences support several subtypes such as strings, lists, and tuples. The other is the *map* type. Maps support associative lookup via a key value. You'll learn about both types in this section.

Sequence types

A *sequence* is an ordered collection of items. All sequences are zero-indexed, which means the first element is element zero (0). Indices are consecutive (that is, 0, 1, 2, 3, ...) to the length (less one) of the sequence. Thus sequences are similar to C and Java arrays.

All sequences support *indexing* (or *subscripting*) to select sub-elements. If *x* is a sequence then the expression `x[n]` selects the *nth* value of the sequence. Mutable sequences such as lists support indexing on assignment, which causes elements to be replaced. For these sequences the expression `x[n] = y` replaces the *nth* element of *x* with *y*.

Sequences support an extension of indexing, called *slicing*, which selects a range of elements. For example, `x[1:3]` selects the second through third elements of *x* (the end index is one past the selection). Like indexing, slicing can be used on assignment to replace multiple elements.

In Jython, a sequence is an abstract concept, in that you do not create sequences directly, only instances of subtypes derived from sequences. Any sequence subtype has all the functions described for sequences.

A slice of life

The many valid forms of slicing are summarized below. Assume *x* is a sequence containing 10 elements (indexes 0 through 9).

Sample expression	Resulting action	Comments
`x[1]`	Selects index 1	Same as indexing
`x[1:2]`	Selects index 1	The end value is one past the selected value
`x[1:]`	Selects index 1 through 9	Missing value implies the sequence length
`x[:7]`	Selects index 0 through 6	Missing value implies zero
`x[:-1]`	Selects index 0 through 8	Negative indexes are adjusted by the sequence length
`x[-6:-3]`	Selects index 3 through 6	Reverse ranges are supported
`x[:]`	Selects index 0 through 9	The whole sequence; This makes a copy of the sequence
`x[:1000]`	Selects index 0 through 9	A reference off the end of the sequence is the end
`x[-100:]`	Selects index 0 through 9	A reference off the start of the sequence is the start
`x[::2]`	Selects index 0, 2, 4, 6, 8	The third value skips over selections

Sequence operators

Jython supports several operations between sequences (*x* and *y*), as summarized below:

Operator	Usage	Example
`x + y`	Join (or concatenate) sequences	`[1,2,3] + [4,5,6]` --> [1,2,3,4,5,6]
`i * x` `x * i`	Repeat sequence	`[1,2,3] * 3` --> [1,2,3,1,2,3,1,2,3]
`o in x` `o not in x`	Contains test	`2 in (1,2,3)` --> 1 (true) `7 not in (1,2,3)` --> 1 (true)

Sequence functions

In addition, several functions can be applied to any sequence (*x*), as summarized below:

Function	Usage	Example
`len(x)`	Length (number of elements) of the sequence	len(1,2,3) --> 3
`min(x)`	Smallest value in the sequence	min(1,2,3) --> 1
`max(x)`	Largest value in the sequence	max(1,2,3) --> 3

A final note about sequences

As I mentioned earlier, a sequence in Jython is an abstract concept, in that you do not create sequences directly, only instances of subtypes derived from sequences. Any sequence subtype has all the functions described for sequences. There are several sequences subtypes, as follows:

- **strings** are immutable sequences of characters (see Strings)

- **tuples** are immutable sequences of any data type (see Tuples)
- **ranges** are immutable sequences of integers (see Ranges)
- **lists** are mutable sequences of any data type (see Lists)

Strings

A *string* is an immutable sequence of characters treated as a value. As such, strings support all of the immutable sequence functions and operators that result in a new string. For example, `"abcdef"[1:4]` is the new string `"bcd"`. For more information on string functions see Appendix B: String methods.

Jython does not have a character type. Characters are represented by strings of length one (that is, one character).

Strings literals are defined by the use of single or triple quoting. Strings defined using single quotes cannot span lines while strings using triple quotes can. A string may be enclosed in double quotes (") or single ones ('). A quoting character may contain the other quoting character un-escaped or the quoting character escaped (proceeded by the backslash (\) character). See Appendix A: Escape characters for more on this.

String examples

Following are some example strings:

- "This is a string"
- 'This is also a string'
- "This is Barry's string"
- 'Barry wrote "Introduction to Jython"!'
- "This is an escaped quote (\") in a quoted string"
- r"\s*xyx\s*" - equivalent to"\\s*xyx\\s"
- u"the number one is \u0031" (vs. "the number one is \x31")

Note that the next-to-last example shows a *raw string*. In raw strings the backslash characters are taken literally (that is, there is no need to double the backslash to get a backslash character). This raw form is especially useful for strings rich in escapes, such as regular expressions. We'll talk more about regular expressions in Part 2 of this tutorial.

The last example shows a *Unicode string* and how to create Unicode escaped values. Note that all strings are stored using Unicode character values (as provided by the JVM); this format just lets you enter Unicode character values.

Mixed and long strings

For convenience, multiple strings separated by only white space are automatically concatenated (as if the `+` operator was present) by the Jython parser. This makes it easy to enter long strings and to mix quote types in a single string. For example the sequential literals here:

```
"This string uses ' and " 'that string uses "."
```

becomes this string:

```
This string uses ' and that string uses ".
```

Triple quoting is used to enter long strings that include new-lines. Strings defined using single quotes cannot span lines while strings using triple quotes can. They can also be used to enter short (single-line) strings that mix quote types. For example, the following is one long multi-line string:

```
r"""Strings literals are defined by the use
single or triple quoting.
Strings defined using single quotes cannot span
lines while strings using triple quotes can.
A string may be enclosed in quotes (") or apostrophes (').
They may contain the other character un-escaped
or the quoting character escaped
(proceeded by the backslash (\) character."""
```

While this is a short mixed-quote string: `'''This string uses ' and that string uses ".'''`

Formatting strings and values

Jython strings supports a special formatting 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)}{width}{.precision}x
```

Here's a guide to the format items:

- **key**: Optional key to lookup in a supplied dictionary
- **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 (in Appendix H: Format codes)

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
```

Tuples

Tuples are immutable lists of any type. Once created they cannot be changed. Tuples can be of any length and can contain any type of object. Some examples are shown here:

Example	Comment(s)
`()`	An empty tuple
`(1,)`	A tuple with one element, an integer; the comma is needed to distinguish the tuple from an expression like (1)
`(1, 'abc', 2, "def")`	A tuple with four elements, two integers and two strings
`((), (1,), (1,2), (1,2,3))`	A tuple of tuples; Each sub-list contains integers
`(1, "hello", ['a','b','c'], "goodbye")`	A mixed tuple of integers, strings and a sub-list of strings
`v1 = 1; v2 = 10` `(1, v1, v2, v1 + v2)`	A tuple of integers; variable references and expressions are supported

Note that although a tuple is immutable, the elements in it may not be. In particular, nested lists (see Lists) and maps (see Maps and dictionaries) can be changed.

Ranges

To implement iteration (see the The for statement) Jython uses immutable sequences of increasing integers. These sequences are called *ranges*. Ranges are easily created by two built-in functions:

- **range({start,} end {,inc})** creates a small range.
  All elements of the range exist.
- **xrange({start,} end {,inc})** creates a large range.
  Elements are created only as needed.

Ranges run from `start` (defaults to 0), up to but not including `end`, stepping by `inc` (defaults to 1). For example:

```
print range(10)        # prints [0,1,2,3,4,5,6,7,8,9]
print range(2,20,2)    # prints [2,4,6,8,10,12,14,16,18]
print range(10,0,-1)   # prints [10,9,8,7,6,5,4,3,2,1]
```

Lists

Lists are mutable sequences of any type. They can grow or shrink in length and elements in the list can be replaced or removed. Lists can be of any length and can contain any type of object. For more information on list functions see Appendix C: List methods. Some examples are shown below.

Example	Comment(s)
`[]`	An empty list
`[1]`	A list with one element, an integer
`[1, 'abc', 2, "def"]`	A list with four elements, two integers and two strings
`[[],[1],[1,2],[1,2,3]]`	A list of lists; Each sub-list contains integers
`[1, "hello", ['a','b','c'], "goodbye"]`	A mixed list of integers, strings and a sub-list of strings
`v1 = 1; v2 = 10` `[1, v1, v2, v1 + v2]`	A list of integers; variable references and expressions are supported

Stacks and queues

Lists support the notion of *Last-In/First-Out* (LIFO) stacks and *First-in/First-out* (FIFO) queues. Using list *x* to create a stack, remove items with `x.pop()` (or the equivalent `x.pop(-1)`). Using list *x* to create a queue, remove items with `x.pop(0)`. To add elements to the list use `x.append(item)`. For example:

```
l = [1,2,3,4,5]  # define a list

l.append(6)      # l is [1,2,3,4,5,6]

w = l.pop()      # w is 6, l is [1,2,3,4,5]
x = l.pop(-1)    # x is 5, l is [1,2,3,4]
y = l.pop(0)     # y is 1, l is [2,3,4]
z = l.pop(0)     # z is 2, l is [3,4]
```

List comprehensions

Lists can also be created via an advanced notation, called list comprehensions. *List comprehensions* are lists combined with `for` and `if` statements to create the elements of the list. For more information see The for statement and The if statement. Some example list comprehensions follow:

Example	Resulting list
`[x for x in range(10)]`	[0,1,2,3,4,5,6,7,8,9] Same as `range(10)`
`[x for x in xrange(1000)]`	[0,1,2,..., 997, 998, 999] Same as `range(1000)`
`[(x < y) for x in range(3) for y in range(3)]`	[0,1,1,0,0,1,0,0,0]
`[x for x in range(10) if x > 5]`	[6,7,8,9]
`[x ** 2 + 1 for x in range(5)]`	[1,2,5,10,17]
`[x for x in range(10) if x % 2 == 0]`	[0,2,4,6,8]

Maps and dictionaries

Mapping types support a mutable set of key-value pairs (called *items*). Maps are distinct from sequences although they support many similar operations. They are similar to sequences in that they are abstract; you work only with map subtypes, of which the most commonly used type is the *dictionary*. For more information on map functions see Appendix D: Map methods.

Maps support associative lookup via the key value. A key can be any immutable type. Keys must be immutable as they are hashed (see Appendix E: Built-in functions) and the hash value must stay stable. Common key types are numbers, strings, and tuples with immutable elements. Values may be of any type (including `None`). If *m* is a map, function `len(m)` returns the number of items in the map.

Maps, like sequences, support subscripting, but by key instead of index. For example, if *m* is a map, `x = m["x"]` gets a value from the map and `m["x"] = x` adds a new value to or replaces a value in the map.

Example dictionaries

Some example dictionary literals are below:

Example	Comment(s)
`{}`	An empty dictionary
`{1:"one", 2:"two", 3:"three"}`	A dictionary with three elements that map integers to names
`{"one":1, "two":2, "three":3}`	A dictionary with three elements that map names to integers
`{"first':'Barry", "mi":"A", "last":"Feigenbaum"}`	A dictionary that maps a name
`{"init":(1,2,3), "term":['x','y','z'], "data": {1:10,2:100.5}}`	A dictionary containing a tuple, a list, and another dictionary
`t = (1,2,3); l = ['x','y','z']; d = {1:10,2:100.5} {"init":t, "term":l, "data":d}`	A dictionary containing a tuple, a list, and another dictionary; variable references and expressions are supported

As shown in Formatting strings and values, dictionaries are convenient for format mapping.

Jython program structure

File structure

As explained in the introduction, Jython programs are simply text files. These files contain statements that are interpreted as they are input (after a quick parsing for syntax errors). Other files can be effectively included into Jython programs by use of the `import` (see Modules and packages) and `exec` statements (see Dynamic code evaluation).

Commentary

Jython has two forms of comments:

- **Remarks** are comments introduced with the sharp (#) character. All text on the same line after the sharp is ignored. Remarks can start in any column.
- **Documentation comments** are a string literal located immediately after the start of an externalized block, such as a module, class, or function. The string does not change the behavior of the block; yet the comment can be accessed via the special attribute `__doc__` to create descriptions of the block.

A commentary example

The following example shows a function (`fac`) that has a documentation comment and two remarks. It also demonstrates how to access the documentation comment programmatically.

The code sequence

```
def fac(x):
     "The fac function computes the value x! (x factorial)"
     if x <= 1: return 1       # base case
     return long(x) * fac(x-1) # use recursion on reduced case
  :
print fac.__doc__
```

results in the output

```
The fac function computes the value x! (x factorial)
```

Statement syntax

As you likely have gathered from the previous sections, Jython has a simple syntax. It more closely resembles English than languages like C and Java language. In particular, each source line is (generally) a single statement. Except for `expression` and `assignment` statements, each statement is introduced by a keyword name, such as `if` or `for`. You may have blank or remark lines between any statements.

You don't need to end each line with a semicolon but you may do so if desired. If you wish to include multiple statements per line, then a semicolon is needed to separate statements.

If required, statements may continue beyond one line. You may continue any line by ending it with the backslash character, as shown below:

```
x = "A looooooooooooooooooooooooooooooooong string " + \
     "another looooooooooooooooooooooooooooooooong string"
```

If you are in the middle of a structure enclosed in parenthesis (()), brackets ([]) or curly braces ({}), you may continue the line after any comma in the structure without using a backslash. Here's an example:

```
x = (1, 2, 3, "hello",
      "goodbye", 4, 5, 6)
```

Identifiers and reserved words

Jython supports identifiers similar to C++ and Java names. *Identifiers* are used to name variables, functions, and classes, and also as keywords. Identifiers can be of any length. They must start with a letter (upper- or lowercase) or the underscore (_) character. They may contain any combination of letters, decimal digits, and the underscore. Some valid identifiers are *abc*, *abc123*, *_x*, *x_*, *myName*, and *ONE*. Some invalid identifiers are *123abc*, *$xyz*, and *abc pqr*.

Note that names starting with underscore are generally reserved for internal or private names.

Jython also has several *reserved words* (or *keywords*) which cannot be used as variable, function, or class names. They fall under the following categories:

- **Statement introducers**: *assert*, *break*, *class*, *continue*, *def*, *del*, *elif*, *else*, *except*, *exec*, *finally*, *for*, *from*, *global*, *if*, *import*, *pass*, *print*, *raise*, *return*, *try*, and *while*.
- **Parameter introducers**: *as*, *import*, and *in*.
- **Operators**: *and*, *in*, *is*, *lambda*, *not*, and *or*.

Note that keywords can be used in special circumstances, such as names of methods. For instance, you might use a keyword to call a Java method with the same name as a Jython keyword. Improper keyword use will generally cause a `SyntaxError`.

Blocks

Blocks (or *suites*) are groups of statements that are used where single statements are expected. All statements that can take a block of statements as a target introduce the block with the colon character. The following statements (or statement clauses) can take a block as their target: *if*, *elif*, *else*, *for*, *while*, *try*, *except*, *def*, and *class*. Either a single statement or small group of statements, separated by semicolons, may follow the colon on the same line, or a block may follow the statement indented on subsequent lines.

I highly recommend that you use spaces to indent. Using tabs can cause problems when moving between systems (or editors) with different tab stops. Do *not* mix tabs and spaces in the same source file. By convention, four spaces are used per level.

Note: All the lines in the outermost block of a module must start at column one; otherwise, a `SyntaxError` is created.

Example blocks

Unlike with C and the Java language, in Jython curly braces are not used to delimit blocks; indentation is used instead. For example

```
# the following prints 0 through 10 on one line
for i in range(10):
     # print next value
     print i,
print  # new line
```

outputs the line: 0 1 2 3 4 5 6 7 8 9.

The block that is the body of the for-loop is indicated by the indented code. All lines in the body (except for comments) must be indented to the same position. The same loop could be written as:

```
# the following prints 0 through 10 on one line
for i in range(10): print i,   # print next value
print  # new line
```

Visibility and scopes

Jython supports the following scopes:

- **Built-in** symbols defined by the Jython runtime are always available unless redefined in another scope.
- **Global** variables are visible to the an entire module, including functions and classes declared in the module. A dictionary of the variables in the current global scope can be accessed via the *globals* function.
- **Local** function arguments and variables declared in a function body are visible to that block. A dictionary of the variable names in the current local scope can be accessed via the *locals* function. In a module and outside of any function, the local and global scopes are the same.

In general, variables are visible in the scope of the block they are declared in and in any function (see Jython functions) defined in that scope. Variables can be declared only once per scope;

subsequent use re-binds that variable. Unlike in C++ and the Java language, nested blocks inside functions do not start new scopes.

Dynamic code evaluation

Jython is distinguished from typical languages in its ability to dynamically create code and then execute it. For example, in a calculator application, the user can enter an expression in text form and Jython can directly execute the expression (assuming it follows Jython source rules).

To better understand how Jython interprets/evaluates dynamic code, consider the following:

```
v1 = 100; v2 = 200
l1 = [1, 2, v1, v2]
d1 = {"simple":123, "complex":(v1, v2, l1)}
expr = raw_input("Enter an expression:")
print eval(expr)    # evaluate and print the expression
```

Below are some sample expressions to evaluate using the code above and the results of those evaluations:

Input expression (entered as a string)	Result
`'1234.56'`	1234.56
`'v1+v2'`	300
`'d1["simple"]'`	123
`'v1**2 + len(l1)'`	10004

Eval, exec and execfile

The `eval` function is used to execute an expression that returns a value. The `exec` statement is used to evaluate a code block (one or more statements) that does not return a value. It takes a file, a string (often read from a file), or a function as its source operand. The `execfile` function executes a code block from a file. In effect it runs a subprogram.

The `exec` statement has the following form:

```
exec source {in globals {, locals}}
```

The `execfile` and `eval` functions have the following form:

```
execfile(filename, {globals {, locals}})

eval(expression, {globals {, locals}})
```

All three forms optionally take two dictionaries that define the global and local namespaces. See Visibility and scopes for more details on namespaces. If these dictionaries are omitted, the current local namespace (as provided by the `locals` function) and the current global namespace (as provided by the `globals` function) are used.

For example, if the dictionaries `gd = {"one":1, "two":2}` and `ld = {"x":100, "y":-1}` are used as namespaces, then this: `print eval("one + two * 2 + x + y", gd, ld)`

prints: `104`.

More details on the use of the `eval` function and `exec` statement are available in the Python Library Reference (see Resources).

Modules and importing

About modules and imports

Jython breaks programs down into separate files, called modules. Modules are reused by importing them into your code. Jython provides many modules for you to reuse (see Appendix F: Jython library summary). Jython also allows you to reuse any Java class and API.

Modules and packages

A *module* is an executable Jython file that contains definitions (for variables, functions and/or classes). Modules are *imported* (executed and bound) into other programs/scripts or modules. It is necessary to import a module when the importing program or module needs to use some or all of the definitions in the imported module.

Jython *packages* are conceptually hierarchically structured sets of modules. They are implemented as directories that contain one or more modules and a special file, `__init__.py`, that is executed before the first module of the package is executed.

Modules and packages enable reuse of the extensive standard Jython and Java libraries. You can also create modules and packages for reuse in you own Jython applications. For more information on the available Jython modules see Appendix F: Jython library summary. For more information on the available Java libraries visit the Sun Microsystems' Java technology home page (in Resources).

The import statement

The `import` statement executes another file and adds some or all of the names bound in it to the current namespace (see Visibility and scopes). The current namespace will generally be the global namespace in the importing file. All statements, including assignments, in the module are executed. The `import` statement comes in several forms:

```
import module {as alias}

   -- or --

from module import name {as alias}

   -- or --

from module import *
```

The `module` value names a Jython (.py) file or dotted-path to a Jython package. The `name` value selects specific names from the module. Module names are case sensitive. These arguments can be repeated. The optional `alias` value allows imported objects to be renamed.

Example imports

Below are some example `import` statements:

Example	Comment(s)
`import sys`	Import the `sys` module. All names in `sys` can be referenced by the prefix *sys*.
`from sys import exc_info`	Imports the `exc_info` function from the `sys` module. No prefix is needed.
`from sys import *`	Imports all the names and functions in the `sys` module. No prefix is needed.
`from sys import exc_info as einfo`	Imports the `exc_info` function from the `sys` module and names it `einfo`. No prefix is needed.
`from string import uppercase as uc, lowercase as lc`	Imports the `uppercase` and `lowercase` functions from module `string`. No prefix is needed.
`import sys, string`	Imports modules `sys` and `string`
`import com.ibm.tools.compiler as compiler`	Imports the `compiler` module from the `com.ibm.tools` package giving it the short name `compiler`.

Importing modules and packages

To import a module or package, Jython must be able to find the associated source (.py) file. Jython uses the `python.path` (very similar to the Java language's `CLASSPATH`) and `python.prepath` variables in the Jython registry to search for these files. You can use any text editor to add to or update the `registry` file in the Jython home directory (usually `c:\jython-2.1`). For more information, see the Jython registry (in Resources) or the `registry` file itself.

By default, Jython will search the directory containing the executing source file; thus, modules located in the same directory as the importing Jython program can be found. Frequently the current directory is also on the path. Simply enter the following command to examine the current search paths:

```
import sys
print sys.path
```

On my machine, when running in the `C:\Articles` directory, the above command produces the following output:

```
['', 'C:\\Articles\\.', 'C:\\jython-2.1\\Lib', 'C:\\jython-2.1']
```

To find Java class files, Jython searches both the Java `CLASSPATH` and the `sys.path` values.

Import is executable

Unlike in the Java language, the `import` statement is executable and is not a compiler directive in Jython. Thus, imports do not need to be done at the start of a module; just sometime before

the imported symbols are used. In fact importing can be done conditionally, as in the following example.

```
   :
# lots of other stuff
   :
if __name__ == "__main__":
     :
     from sys import exit
     exit(0)
```

Imports can also be undone, as shown here:

```
import sys
   :
# lots of other stuff
   :
del sys
```

Subsetting imports

When you import modules, all values assigned or functions created in the module are usually available for reference by the module importer. You can prevent this by altering the code within the module. Either start the name with an underscore (_) or define a special variable, `__all__`, at the start of the module, listing only the names of the variables or functions you want to be imported. For example, the `__all__` definition below:

```
__all__ = ["getline","clearcache","checkcache"]
```

would only import the names `getline`, `clearcache`, and `checkcache`.

A similar strategy can be used at the module directory level. Defining the variable `__all__` in a file called `__init__.py` instructs the interpreter as to which modules to import from the package if the wildcard (*) is used in the import statement. For instance, if the line `__all__ = ['mod1', 'mod3', 'globals']` is in a file called `__init__.py` in a directory named *modules*, it will cause the statement `from modules import *` to import the modules `mod1`, `mod3`, and `globals` from the *modules* directory.

Running native applications

Using the `os.system` function, Jython can also run any external program that can be found on the current host `PATH`, such as a host operating system application. For example, to compile a Java program you could use

```
import os
import sys

cmd = "javac %(name)s.java 1>%(name)s.out 2>%(name)s.err" % \
                  {'name': sys.argv[1]})
rc = os.system(cmd)
if rc == 0:
     print "Successful"
else:
     print "Failed: return code=%i..." % rc
     # read and process the .err file...
```

Jython exceptions

About exceptions

Regardless of how much care a programmer takes in designing and testing his or her code, unexpected errors, or *exceptions*, can occur. Jython provides excellent support for recovering from these errors,

Exceptions are generally subclasses of the Jython type `exceptions.Exception` or the Java class `java.lang.Exception`. Most Jython exception names end in "Error" (such as `IOError` or `IndexError`) or "Warning." Java exceptions end in either "Error" (for critical exceptions) or "Exception" (for generally recoverable exceptions). For more information see The Jython exception hierarchy or the Python Library Reference (see Resources for a link).

The Jython exception hierarchy

Here is Jython's principle exception hierarchy subset.

- 1 Exception
  - 1.1 SystemExit
  - 1.2 StopIteration
  - 1.3 StandardError
    - 1.3.1 KeyboardInterrupt
    - 1.3.2 ImportError
    - 1.3.3 EnvironmentError
      - 1.3.3.1 IOError
      - 1.3.3.2 OSError
    - 1.3.4 EOFError
    - 1.3.5 RuntimeError
      - 1.3.5.1 NotImplementedError
    - 1.3.6 NameError
      - 1.3.6.1 UnboundLocalError
    - 1.3.7 AttributeError
    - 1.3.8 SyntaxError
      - 1.3.8.1 IndentationError
      - 1.3.8.2 TabError
    - 1.3.9 TypeError
    - 1.3.10 AssertionError
    - 1.3.11 LookupError
      - 1.3.11.1 IndexError
      - 1.3.11.2 KeyError
    - 1.3.12 ArithmeticError
      - 1.3.12.1 OverflowError
      - 1.3.12.2 ZeroDivisionError
      - 1.3.12.3 FloatingPointError

        - 1.3.13 ValueError
        - 1.3.14 ReferenceError
        - 1.3.15 SystemError
        - 1.3.16 MemoryError
- 2 Warning
    - 2.1 UserWarning
    - 2.2 DeprecationWarning
    - 2.3 PendingDeprecationWarning
    - 2.4 SyntaxWarning
    - 2.5 OverflowWarning
    - 2.6 RuntimeWarning
    - 2.7 FutureWarning

This hierarchy is a subset of the Python Library Reference (see Resources). These exceptions may be subclassed.

The try-except-else statement

Like C++ and the Java language, Jython supports exception handlers. These handlers are defined by the `try-except-else` statement, which has the following form:

```
try: statement
except type, var: statement
  :
else: statement

-- or --

try:
    block
except type, var:
    block
  :
else:
    block
```

The `except` clause may be repeated with different `type` values. If so, the exceptions either must not overlap hierarchically (that is, be siblings) or they must be ordered from child to root exceptions. The optional `type` value is an exception type (either a subclass of `exceptions.Exception` or `java.lang.Throwable`). If `type` is missing, then the `except` clause catches all Jython and Java exceptions. The optional `var` value receives the actual exception object. If `var` is missing, then the exception object is not directly accessible. The `else` clause is optional. It is executed only if no exception occurs.

If an exception occurs in the `try` clause, the clause is exited and the first matching `except` clause (if any) is entered. If no exception matches, the block containing the `try-except-else` is exited and the exception is re-raised.

If an exception is raised in the `except` or `else` clause, the clause will exit and the new exception will be processed in the containing block.

Accessing exception information

To access information about an exception, you may use the value provided in the `except` clause as described previously or the `sys.exc_info` function. For example, you can use the following function, in which `type` is the class of the exception, `value` is the exception object (use `str(value)` to get the message), and `traceback` is the execution trace back, which is a linked list of execution stack frames.

```
import sys
  :
try:
     :
except:
     type, value, traceback = sys.exc_info()
```

More details on the exceptions and trace backs is available in the *Python Reference Manual* (see Resources).

The try-finally statement

Like C++ and the Java language, Jython supports an additional construct, `try-finally`, which makes it easy to do required cleanup activities such as closing open files, releasing resources, etc. Any code in the `finally` clause is guaranteed to be executed once the `try` clause is entered, even if it is exited via a return statement (see The return statement) or an exception. The `try-finally` statement has the following forms:

```
try: statement
finally: statement

  -- or --

try:
    block
finally:
    block
```

Note that `try-except-else` statements may nest in `try-finally` statements and vice versa.

A try statement example

Here is an example of using both `try-except` and `try-finally` statements together. We'll talk more about Jython file I/O in Part 2 of this tutorial.

```
def readfile (name):
     "return the lines in a file or None if the file cannot be read"
     try:
         file = open(name, 'r')  # access the file
         try:
             return file.readlines()
         finally:
             file.close()    # ensure file is closed
     except IOError, ioe:    # report the error
         print "Exception -", ioe

:

# prints Exception - File not found - nofile (...)
# then None
print readfile("nofile")

# prints a list of the lines in the file
print readfile("realfile")
```

The raise statement

Exceptions are generated by called functions or built-in services. You can also generate one by using the `raise` statement. The `raise` statement has the following forms:

```
raise exception

  -- or --

raise exception_class {, message}

  -- or --

raise
```

Below are some example `raise` statements.

Example	Comment(s)
`raise`	Re-raise the current exception; used in an except block to regenerate the exception
`raise IOError`	Create and raise an `IOError` with no message
`raise anIOError`	Re-raise an existing `IOError` object
`raise IOError, "End of File"`	Create and raise an `IOError` with a explanatory message
`from java import io` `raise io.IOException, "End of File"`	Create and raise a Java exception with a explanatory message

Jython procedural statements

A statement for every procedure

Jython has a number of statements that perform computation or control program flow, including the `expression`, `assignment`, `pass`, `if`, `while`, `for`, `break`, `continues`, and `del` statements. You'll learn about these procedural statements in the sections that follow.

The pass statement

The `pass` statement is used where a Jython statement is required syntactically but when no action is required programmatically. `pass` can be useful to create empty loops or to provide a temporary implementation of a block. The statement has the following form:

```
pass
```

The expression statement

In Jython, any expression can serve as a statement; the resulting value is simply discarded. Most often any such `expression` statement calls a function or method (discussed further in Part 2). For example, the following code invokes three functions in sequence:

```
performAction(1)
performAction(2)
performAction(3)
```

Operators and precedence

Jython expressions consist of any valid combination of the operators described in Summary of operator precedence. They are similar to the expressions of most languages, especially C/C++ and the Java language.

```
1 + 1               # add 1 and 1 yielding 2

(1 + 2) ** 3        # add 1 and 2 and raise the result by 3 yielding 27

1 + 2 ** 3          # raise 2 by 3 and add 1 yielding 9

x % y == 0          # tests to see if x is divisible by y

x & 1               # extracts the low-order bit of x

# below is the same as: "(0 <= x) and (x < 100)" but is more concise
0 <= x < 100        # tests a range

# the use of (...) below is not required but it improves readability
(x > 0) and (y < 0)  # tests the relation of 2 values

1 + 2 * 3 ** 4 << 2  # complex expression yielding 652
(1 + (2 * (3 ** 4))) << 2   # Equivalent fully parenthesized expression
```

Summary of operator precedence

Jython operator precedence is summarized in the table below. Use parentheses to change the order or to improve readability. Unless otherwise noted, within the same precedence level operations are evaluated left-to-right. Higher priority operations are at the top of the list.

Operation	Comment
`( expression )`	Nested expression or grouping
`( expr1, ..., exprN )`	Tuple constructor
`[ expr1, ...,exprN ]`	List constructor
`{ key1:value1, ..., keyN:valueN }`	Dictionary constructor

`` `expression` ``	repr (representation) expression

`x.name`
`x[i], x[i:j], x[i:j:k]`
`x(...)`
	Member (attribute or method) selection
	Subscripting or slicing
	Function call

`**`	Raise to power (right associative)

`+`
`-`
`~`
	Posate
	Negate
	Bit-wise not

`*`
`/`
`%`
	Times
	Divide
	Modulo

`+` `-`	Plus Minus
`<<, >>`	Bit-wise shifts
`&`	Bit-wise and
`^`	Bit-wise xor
`|`	Bit-wise or

`is`, `is not`
`in`, `not in`
`<`, `<=`, `>`, `>=`, `==`, `!=`, `<>`
	Sameness test
	Containment test
	Relational test

`not`	Logical not
`and`	Logical and
`or`	Logical or
`lambda`	Declare a lambda function

The assignment statement

The `assignment` statement is used to bind (or re-bind) a value to a variable. If not already defined, *binding* creates the variable and assigns it the value. In Jython all data are objects, so variables actually store references to objects, not values. Variables are not typed, thus an assignment can change the type of the value a variable references.

More than one variable can have a reference to the same object; this is called *aliasing*. For this reason, Jython supports the `is` and `is not` operators to test whether or not two variables refer to the same object.

A variable can only be declared once in a block. This means that it is declared (by the parser) in the block even if the flow in the block does not execute the `assignment` statement that creates it. The variables will have an undefined value until the first assignment is actually executed.

Note that like the `assignment` statement other statements can bind variables. Some examples are the `class`, `def`, `for`, and `except` statements.

Parallel assignment

The `assignment` statement supports sequence unpacking. This can achieve a form of parallel assignment. For example, the following sets *a* to *1*, *b* to *2*, and *c* to *3*:

```
(a, b, c) = (1, 2, 3)

-- or --

x = (1, 2, 3)
(a, b, c) = x
```

The same number of variables must be on the left side as on the right side. This unpacking can be very useful if you are provided with a sequence variable (say as an argument to a function) and want to access the values within it. For convenience, the enclosing parentheses are not required, so the above assignment could also be written as `a, b, c = 1, 2, 3`.

Multiple assignment

Jython supports the use of multiple assignment. For example, the following sets *c* to *1*, *b* to *c* (or *1*), and *a* to *b* (also *1*).

```
a = b = c = 1
```

Thus, `assignment` is unusual in that it is right-associative.

Augmented assignment

Jython supports augmented assignment, which combines operators with `assignment`. The general form is `v <op>= expression`, which is equivalent to `v = v <op> expression`, except that `v` is evaluated only once (which can be important in a subscripted variable).

The following augmented assignment operators are available:

```
+=
-=
*=
/=
%=
**=
<<=
>>=
&=
|=
^=
```

The if statement

The `if`, `elif`, and `else` statements provide basic decision capability. The test expressions evaluate to false (None, 0 or empty) or true (not-0 or not-empty).

This form is used to execute a statement or block conditionally:

```
if expression: statement

  -- or --

if expression:
    block
```

Here's an example:

```
if x < 0: x = -x  # ensure x is positive
```

The if-else statement

The following form is used to choose between two alternative statements and/or blocks:

```
if expression: statement
else:          statement

  -- or --

if expression:
    block
else:
    block
```

Here's an example:

```
if x >= 0:
     result = fac(x)
else:
     print x, "is invalid for a factorial"
```

The if-elif-else statement

The following form is used to choose between a set of alternative statements and/or blocks:

```
if   expression: statement
elif expression: statement
:
else:            statement

  -- or --

if   expression:
    block
elif expression:
    block
:
else:
    block
```

The `elif` clause can repeat. The `else` clause is optional. Here's an example:

```
if   x == 0:
     doThis()
elif x == 1:
     doThat()
elif x == 2:
     doTheOtherThing()
else:
     print x, "is invalid"
```

Conditional expressions

Most languages based on C, including C++ and the Java language, support a *conditional expression*. These expressions return a choice of sub-expressions. They are especially useful to avoid the use of repeated targets. This is important if the target contains complex expressions, such as subscripts. Conditional expressions have the form

```
target = expression ? true_expression : false_expression
```

Jython does not support conditional expressions directly; instead it employs this form:

```
if expression: target = true_expression; else: target = false_expression

-- or --

if expression: target = true_expression
else:          target = false_expression
```

You can approximate the Java conditional expression form in Jython using the `and`, and `or` operators, as shown here:

```
target = (expression) and (true_expression) or (false_expression)
```

Note that this form works only if `true_expression` and `false_expression` do not themselves evaluate to false values (such as `None`, 0, or an empty sequence or map). If that is the case, use the `if-else` form.

Implementing a switch statement

Jython does not support a `switch` or `case` statement like many other languages do. The `if-elif-else` form can be used to do similar tests for a limited number of cases. For more cases, you can use sequences or maps to functions (see Jython functions), as follows:

```
:
# define the function to handle each unique case
def case1(...): return ...
def case2(...): return ...
   :
def caseN(...): return ...

# defines cases in a dictionary; access by key
cases = {key1:case1, key2:case2, ..., keyN:caseN}
:
result = cases[key](...)  # get the selected case result

-- or --

# defines cases in a list or tuple; access by index
cases = (case1, case2, ..., caseN)
:
result = cases[x](...)  # get the xth case result
```

The while statement

The `while` statement is used to perform conditional looping. As long as the expression evaluates to *true* the loop is executed. The `while` statement has the following forms:

```
while expression: statement
else:             statement

  -- or --

while expression:
    block
else:
    block
```

The `else` clause, which is optional, is executed only if the `while` clause ends normally (that is, not with a `break` statement). It is not typically used.

Example:

```
x = 10
while x > 0:
     process(x)
     x -= 1
```

The following is an infinite loop:

```
while 1: print "Hello!"
```

The for statement

The `for` statement is used to perform iterative looping. It processes a sequence returned from the supplied `expression`, taking each element in turn. As long as elements remain in the sequence, the loop is executed. The `for` statement has the following forms:

```
for vars in expression: statement
else:                   statement

  -- or --

for vars in expression:
    block
else:
    block
```

The `else` clause, which is optional, is executed only if the `for` clause ends normally (that is, not with a `break` statement). It is not typically used.

Example:

```
for c in "String":
     processCharacter(c)  # process the chars in turn

   -- or --

values = (1,2,5,7,9,-1)
for v in values:
     process(v)  # process the values supplied

   -- or --

for i in range(10):
     print i    # print the values 0 to 9 on separate lines

   -- or --

for k, v in {"one":1, "two":2, "three":3}.items():
     print k, '=', v    # print the keys and values of a dictionary
```

In the above code, the use of `for i in range(limit)` provides for the typical *for-loop* or *do-loop* iteration of most languages. Note also that the `keys` will not necessarily come out in the order supplied.

The break and continue statements

It is sometimes necessary to exit a `while` or `for` loop in the middle. Often this is the result of some unusual condition that cannot be tested in the loop mechanism itself. The `break` statement provides this behavior. The statement has the following form:

```
break
```

Here's an example:

```
for i in range(100):
     d = getData(i)
     if not valid(d): break    # can't continue
     processData(d)
```

Likewise, it may be occasionally necessary to skip the rest of the loop body and begin the next iteration early. The `continue` statement provides this behavior. The statement has the following form:

```
continue
```

Here's an example:

```
for i in range(100):
     :     # *** some preliminary work ***
     if not valid(i): continue          # can't process this one
     :     # *** some other stuff ***
```

In general, the `break` and `continue` statements occur as the target of an `if` statement. Only the most closely contained loop can be controlled.

Updating a sequence

You must take special care when updating (that is, inserting or deleting entries from) a sequence while iterating over it, as this can cause the iteration indexes to change unpredictably. I recommend you make a copy of the sequence to iterate over, as shown below:

```
for i in x[:]:    # iterate over a copy
     if i > 0:     # remove all positive values
         x.remove(i)
```

The del statement

Mutable sequences (see Sequence types), maps (see Maps and dictionaries), and classes support the `del` statement, which removes an element or attribute from the collection or class, respectively. For sequences, removal is by index; for maps it is by key value; and for classes it is by attribute name. We'll talk more about class support for the `del` statement in Part 2 of this tutorial.

Local or global variables can be deleted; this removes the variable from the namespace (it does not delete the object the variable refers to). The `del` statement also supports the slice notation.

Assuming "l" is the list [1,2,3,4,5], *d* is the dictionary {1:"one", 2:"two", 3:"three"} and *x* is some class instance, some example `del` statements are as follows:

Example	Comment(s)
`del l[0]`	Removes the first element
`del l[1:3]`	Removes the second through third elements
`del l[::2]`	Removes the even elements
`del l[:]`	Removes all the elements
`del d[1]`	Removes the element with key *1*
`del x.attr1`	Removes the attribute `attr1`

`var = [1,2,3]`
`:`
`del var`
	Removes the variable `var` from its namespace

Jython functions

What are functions?

One of the most important features of any programming language is code reuse. There are two principle methods of code reuse: blocks of code that return values (called *functions*) and blocks that do not return values (called *subroutines*). Jython fully supports code reuse via functions.

Jython provides many built-in and library functions for you to use (see Appendix E: Built-in functions and Appendix F: Jython library summary). Built-in functions can be used without importing them; library function must first be imported.

Jython goes beyond many languages, including the Java language, by making functions *first-class* objects that can be manipulated just like other objects (and most specifically, objects that can be stored in collections and passed as arguments to other functions).

The def statement

In Jython, functions are declared by the `def` statement, which has the following form:

```
def name (  args  ): statement

  -- or --

def name ( args ):
    block
```

Within a given scope (module, function, or class), each function name should be unique. The function name is really a variable bound to the function body (similar to any other assignment). In fact, multiple variables can be defined to reference the same function. The function body can be a single (frequently a return) statement or (more commonly) a block of statements.

Specifying function arguments

The optional *args* in the `def` statement is a comma-separated list of argument definitions. Some examples follow:

Example	Comment(s)
`def x(a, b, c)`	Defines a function with three required positional arguments.
`def x(a, b, c=1)`	Defines a function with three arguments, the last of which is optional with a default value of 1.
`def x(a=3, b=2, c=1)`	Defines a function with three arguments, all of which are optional and have default values.
`def x(p1, p2, kw1=1, kw2=2)`	Defines a function with two positional parameters and two keyword (optional) parameters. When declared, all optional (=value) parameters must follow all non-optional parameters. When this function is called, the keyword parameters, if provided, can be specified by name and in any order after positional parameters.
`def x(p1, p2, *v)`	Defines a function with two required positional parameters and an indeterminate number of variable parameters. The `v` variable will be a tuple.
`def x(p1, p2, **kw)`	Defines a function with two required positional parameters and an indeterminate number of keyword parameters. The `kw` variable will be a dictionary.
`def x(p1, p2, *v, **kw)`	Defines a function with two required positional parameters and an indeterminate number of positional and keyword parameters. The `v` variable will be a tuple. The `kw` variable will be a dictionary.

Handling mutable arguments

If an argument accepts a default value of a mutable object (such as a list or dictionary) it's best to use the this form:

```
def x(p1, p2, p3=None)
     if p3 is None: p3 = []
     :
```

rather than the simple one:

```
def x(p1, p2, p3=[])
    :
```

Because the value after the equals sign is evaluated only when the function is defined (not each time it is called), the list in the second example above will be shared across all calls to the function. This is usually not the desired behavior. The first example gets a distinct list for each call.

Anonymous functions

You can define anonymous functions, called `lambda` functions. Anonymous functions are one-line functions that are typically used as arguments to other functions. These functions are declared using the following form:

```
lambda args: expression
```

The args list is the same as the one described in Specifying function arguments. You should have at least one argument per `lambda` function. The expression value is returned by the function.

The return statement

Functions return values via the `return` statement, which also exits a function. The `return` statement may or may not return an explicit value; if no value is specified, then `None` is used. If the last statement of a function body is not a `return` statement, then a value-less return is assumed. The `return` statement has the following forms:

```
return expression

  -- or --

return
```

Alternatively, this form lets you return multiple values as a tuple:

```
return expression1, expression2, ..., expressionN
```

Function calls

Functions are called by use of the `call` operator, which is a parenthesized list following a function reference. For example, if *f* is a function, then `f(...)` calls the function. If the function definition supports arguments, then the call may include parameters, as shown in the examples below:

Function definition	Example call(s)	Comment(s)
`def x(a, b, c)`	`x(1,2,3)` `x("1", "2", "3")`	Actual arguments can be of any type.
`def x(a, b, c=1)`	`x(1,2,3)` `x(1,2)`	Parameter `c` can be omitted

`def x(a=3, b=2, c=1)`	`x()`
	`x(1,2,3)`
	`x(c=10, a="d")`
	`x(1,c=5)`
		Named parameters can be treated as positional, keyword, or mixed. If keyword, order is not important.

`def x(p1, p2, kw1=1, kw2=2)`	`x(1,2)` `x(1,3,kw2=5)`	Both positional and keyword parameters can be used.

`def x(p1, p2, *v)`	`x(1,2)`
	`x(1,2,3)`
	`x(1,2,3,4)`
		The `v` tuple gets the third and subsequent parameters.

`def x(p1,p2,**kw)`	`x(1,2, aaa=1, mmm=2, zzz=3)`	Keywords can have any name.
`def x(p1,p2,*v, **kw)`	`x(1,2,3,4, aaa=1, xxx="yyy")`	The `v` tuple gets the third and subsequent positional parameters while the dictionary `kw` gets the `aaa` and `xxx` keys with values.

Note that spaces are optional between parameter declarations and arguments. Adding a space between each is recommended for increased readability.

Example function definitions and calls

Below are some example function definitions with example calls.

```
def k(): return 1   # return the constant 1
print k()           # prints 1

# below replaces the built-in abs definition
def abs(x):         # calc the absolute value
     if x < 0: return -x
     return x
print abs(-10)      # prints 10

sum = lambda x, y: x + y   # define sum function
print sum(1,2)      # prints 3

prod = lambda x, y: x * y  # define prod function
print prod(1,2)     # prints 2

# fully equivalent to the above,
# but the above binding of prod is replaced
def prod(x, y): return x * y      # define the prod function
print prod(1,2)     # prints 2

# make an alias of prod
xy = prod
print xy(1,2)       # prints 2

# a function that takes a function
#  similar to the built-in function apply
def applyer (func, args):
     return func(*args)

print applyer(xy, (1,2))                  # prints 2
print applyer(lambda x,y: x ** y, (2,16))  # prints 65536

def factorial(x):
     "calculate the factorial of a number"
     if x < 0:
         raise ValueError, "negative factorial not supported"
     if x < 2:
         return 1
     return long(x) * factorial(x-1)

print factorial(3)              # prints 6
```

The global statement

Occasionally, you may want to declare (that is, assign to) a variable in a local context (such as in a function) but reference a variable in the global scope. To do this, use the `global` statement before the first use of the variable. Here's an example:

```
x = 10; y = 20; z = 30  # three global variables

def f1(p, q, r):
     x = p      # local x, y & z variables
     y = q
     z = r

def f2(a, b, c):
     global x, y
     x = a      # global x & y variables
     y = b
     z = c      # local z variable

print x, y, z   # prints: 10, 20, 30

f1(1, 2, 3)
print x, y, z   # prints: 10, 20, 30

f2(-1, -2, -3)
print x, y, z   # prints: -1, -2, 30
```

Note that as long as it is not re-bound locally, a global variable can be read without first declaring it to be a global. Thus the `global` statement is only required to assign to a global variable.

Generic functions

Similar to in Smalltalk functions, Jython functions are *generic* in that any type can be passed in for each argument. This makes functions extremely flexible. Generic functions work as long as the operations performed on the arguments in the function are valid for the argument's actual type. For example, with these functions' definitions

```
def sum (x, y):
     return x + y

def prod (x, y):
     return x * y
```

the following function calls are valid:

```
print sum(1, 2)             # prints 3

print sum('123', '456')     # prints 123456

print sum([1,2,3], [4,5,6]) # prints [1,2,3,4,5,6]

print prod(2, 3)            # prints 6

print prod('123', 2)        # prints 123123
```

Dynamic type testing

You can use dynamic type testing (that is, using the `isinstance` function or comparing the results of the `type` function) for even more flexibility. See Appendix G: Jython types summary for more information. Here's an example:

```
# See Part 2 of this tutorial for the definition of the UserList class
from UserList import *
  :
data = None     # undefined until setData called
prevdata = []

def setData (values=None):
     """ Set global data. """
     global data, prevdata                # use the global data
     if not data is None:                 # save any prior versions
         prevdata.append(data)
     data = []                            # create default empty data space
     if not values is None:               # some initial values supplied
         if isinstance(values, UserList):  # values is a UserList
             data = values.data[:]         # set to copy of UserList's values
         else:                            # values is some other sequence
             # this will fail if values is not some form of sequence
             data = list(values)          # convert to a list
  :
print data      # prints None
  :
setData("123456")
print data      # prints ['1','2','3','4','5','6']
  :
setData((1,2,3,4,5,6))
print data      # prints [1,2,3,4,5,6]
  :
xdata = data[:]; xdata.reverse()
setData(xdata)
print data      # prints [6,5,4,3,2,1]
  :
print prevdata  # prints [['1',2','3','4','5','6'],[1,2,3,4,5,6]]
```

Nested functions

Unlike many other languages, including the Java language, Jython allows functions to be defined inside of other functions. The nested (or *local*) functions can help to reduce the scope of functions. Here's an example:

```
def x(a, b, c):
     y = a * b

     def square(x):
         return x ** 2      # this x is different from function x

     y *= square(c)
     return x
```

The nested function has no visibility into the variables in the containing function. If the nested function must use these values, pass them into the function as arguments. For example, the following function

```
def calc(a, b, c):
     x = a * b * c

     def sum(data):
         # cannot access calc's namespace (x, a, b, c, or sum) here
         print locals()
         return data['a'] + data['b'] + data['c']

     x += sum(locals())
     return x

print calc(10,20,30)
```

prints

```
{'data': {'x': 6000, 'c': 30, 'b': 20, 'sum': \
     <function sum at 32308441>, 'a': 10}}
6060
```

Nested functions can also be used to conveniently create (preconfigured) functions to return as a result, as shown here:

```
def makeSq(n):
     def sq(x=n):  # n's value is saved as the parameter x value
         return x ** 2
     return sq
```

The above function can be used like this:

```
sq2 = makeSq(2)
print "2*2=%i" % sq2()     # prints: 2*2=4

sq10 = makeSq(10)
print "10*10=%i" % sq10()  # prints: 10*10=100
```

Functional programming

Like Lisp and Smalltalk, Jython supports a limited form of functional programming. Functional programming uses the first-class nature of Jython functions and performs operations on functions and data structures. The built-in functional programming services are shown below:

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

`map(func, list, ...)`	Creates a new list from the results of applying `func` to each element of each list. There must be one list per argument to the function.	`map(lambda x, y: x+y, [1,2],[3,4])`
		`--> [4,6]`
		`map(None, [1,2],[3,4]) --> [[1,3], [2,4]]`

`reduce(func, list {,init})`	Applies `func` to each pair of items in turn. The results are accumulated.	`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`

`filter(func, seq)`	Creates a new list from `seq` selecting the items for which `func` returns *true*. `func` is a one-argument function.	`filter(lambda x: x>0, [1,-1,2,0,3]) --> [1,2,3]`

Using functions like `map`, `reduce`, and `filter` can make processing sequences (that is, strings, lists, and tuples) much easier. These functions are *higher-order* functions because they either take functions as arguments or return them as results.

Functional programming examples

We'll close this section on Jython functions, and the first half of the "Introduction to Jython" tutorial, with some functional programming examples.

A factorial calculator can be implemented using `reduce`:

```
def fac(x):
    return reduce(lambda m,n: long(m)*n, range(1,x))

print fac(10)      # prints 362880L
```

List modification can be done using `map`:

```
l = [1, -2, 7, -3, 0, -11]
l = map(abs, l)

print l      # prints [1, 2, 7, 3, 0, 11]
```

A set of functions can be executed in a sequence using `map`:

```
def f1(x): return ...
def f2(x): return ...
def f3(x): return ...
  :
def fM(x): return ...
  :
def fN(x): return ...

# x=(5) is an example of a parameter for each function,
#   any expression is allowed, each function will get it
# the list determines the order of the functions
# the result of each function is returned in a list.
results = map(lambda f,x=(5): f(x), [fN,f3,f2,f3,...,fM,...,f1])
```

Looping can be achieved using `map`:

```
def body1(count):
    # any body here
    :

# do body 10 times, passing the loop count
map(body1, range(10))

def body2(x,y,z):
    # any body here
    :

# do body with multiple parameters
#   calls body2(1, 'a', "xxx")
#   then  body2(2, 'b', "yyy")
#   then  body2(3, 'c', "zzz")
map(body2, [1,2,3], "abc", ["xxx", "yyy", "zzz"])
```

Selection can be achieved using `filter`:

```
# Assume a class Employee exists with attributes
#  name, age, sex, title, spouse and children (among others)
#  and that instances such as John, Mary and Jose exist.
# See Part 2 of this tutorial for more information on using classes.

John = Employee('John', 35, 'm', title='Sr. Engineer')
Mary = Employee('Mary', 22, 'f', title='Staff Programmer')
Jose = Employee('Jose', 50, 'm', title='Dept. Manager', children=[...])
employees = [John, Jose, Mary]
```

Here's an example of how we'd use the above `filter` to select some employees:

```
# returns: [Jose]
hasChildren = filter(lambda e: e.children, employees)

# returns: []
over65 = filter(lambda e: e.age>65, employees)

# returns: [Mary]
isProgrammer = filter(lambda e: \
                 e.title and e.title.lower().find('prog') >= 0, employees)
```

Wrap-up

Summary

In this first half of the two-part "Introduction to Jython" tutorial, you've learned the concepts and programming basics of working with Jython, including access options and file compilation, syntax and data types, program structure, procedural statements, and functional programming with Jython.

In the second half of this tutorial, we will begin to wrestle with some of the more advanced aspects of the language, starting with a conceptual and hands-on introduction to object-oriented programming in Jython. You'll also learn about debugging, string processing, file I/O, and Java support in Jython. The tutorial will conclude with an exciting, hands-on demonstration of how to build a working GUI app in Jython.

It's a good idea to take the second part of the tutorial as soon as you can, while the concepts from Part 1 are still fresh in your mind. If you prefer to take a break in your studies, you might want to use the time to explore the appendices included with Part 1 (Appendices), or check out some of the references included in the Resources section.

Appendices

Appendix A: Escape characters

Several special characters have backslash versions:

Backslash Representation	Character
`\t`	Tab
`\v`	Vertical-Tab
`\n`	New-Line
`\r`	Return
`\f`	Form-Feed
`\"`	Quote
`\'`	Apostrophe
`\\`	Backslash
`\b`	Backspace
`\a`	Bell
`\000`	Octal value (3 base-8 digits in range 0-377_8)
`\xXX...`	Hex value (2 base 16-digits in range 0-FF_{16}) used in strings (that is, "\x31" --> '1')
`\uXXXX...`	Hex value (4 base 16-digits in range 0-$FFFF_{16}$); used in unicode strings (that is, u"\u0031" --> '1')

Appendix B: String methods

Strings support several useful methods:

Method	Usage	Example
`s.capitalize()`	Initial capitalize s	"abc".capitalize() --> "Abc"
`s.count(ss {,start {,end}})`	Count the occurrences of ss in s[start:end]	"aaabbccc".count("ab") --> 1
`s.startswith(str {, start {, end}})` s.endswith(str {, start {, end}})	Test to see if s starts/ends with str	"xxxyyyzzz".startswith("xx") --> 1
`s.expandtabs({size})`	Replace tabs with spaces, default size: 8	"x\ty".expandtabs(4) --> "x y"
`s.find(str {, start {, end}})` `s.rfind(str {, start {, end}})`	Finds first index of str in s; if not found: -1, rfind searches right-to-left	"12345".find('23') --> 1
`s.index(str {, start {, end}})` `s.rindex(str {, start {, end}})`	Finds first index of str in s; if not found: raise ValueError. rindex searches right-to-left	"12345".index('23') --> 1
`s.isalnum`	Test to see if the string is alphanumeric	"12345abc".isalnum() --> 1

`s.isalpha`	Test to see if the string is alphabetic	"12345abc".isalpha() --> 0
`s.isnum`	Test to see if the string is numeric	"12345abc".isnum() --> 0
`s.isupper`	Test to see if the string is all uppercase	"abc".isupper() --> 0
`s.islower`	Test to see if the string is all lowercase	"abc".islower() --> 1
`s.isspace`	Test to see if the string is all whitespace	"12345 abc".isspace() --> 0
`s.istitle`	Test to see if the string is a sequence of initial cap alphanumeric strings	"Abc Pqr".istitle() --> 1

`s.lower()`
`s.upper()`
`s.swapcase()`
`s.title()`
	Convert to all lower, upper, opposite, or title case	"abcXYZ".lower() --> "abcxyz"
		"abc def ghi".title() --> "Abc Def Ghi"

`s.join(seq)`	Join the strings in seq with s as the separator	" ".join(("hello", "goodbye") --> "hello goodbye"
`s.splitlines({keep})`	Split s into lines, if keep true, keep the newlines	"one\ntwo\nthree".splitlines() --> ["one", "two", "three"]
`s.split({sep {, max}})`	Split s into "words" using sep (default of white space) for up to max times	"one two three".split() --> ["one", "two", "three"]

`s.ljust(width)`
`s.rjust(width)`
`s.center(width)`
`s.zfill(width)`
	Left, right or center justify the string in a field width wide. Fill with 0.	"xxx".rjust(8) --> " xxx"
		"xxx".center(8) --> " xxx "
		str(10).zfill(10) --> "0000000010"

`s.lstrip()`
`s.rstrip()`
`s.strip()`
	Remove leading (and/or trailing) white space	" xxx ".strip() --> "xxx"

`s.translate(str {,delc})`	Translate s using table, after removing any characters in delc. str should be a string with length == 256	"ab12c".translate(reversealpha, "0123456789") --> "cba"
`s.replace(old, new {, max})`	Replaces all or max occurrences old string old with string new	"11111".replace('1', 'a', 2) --> "aa111"

Note: other methods are supported, for a complete list see the Python Library Reference (Resources). Also note that by including the string module, many (but not all) of these methods can also be called as functions, i.e.- `string.center(s, 10)` is the same as `s.center(10)`.

The string module has some important variables:

Variable	Comment(s)
digits octdigits hexdigits	The decimal, octal, and hexadecimal digits
lowercase uppercase letters	The lowercase alphabet, the uppercase alphabet, and the union of them
whitespace	The legal white space characters

Appendix C: List methods

Lists support several useful methods.

Function	Comment(s)	Example

x in l x not in l	Test for containment	1 in [1,2,3,4] --> 1
l.count(x)	Count the occurrences of x. Uses "==" to test.	[1,2,3,3].count(3) --> 2
l.append(x) -- or -- l = l + [x]	Append x to the list	[1,2].append([3,4]) --> [1,2,[3,4]] [1,2] + [3] --> [1,2,3]
l.extend(list)	Appends the elements of list	[1,2].extend([3,4]) --> [1,2,3,4]
l.index(item)	Finds the index of item in list; if not present, raise ValueError	[1,2,3,4].index(3) --> 2
l.insert(index, x) -- or -- l[i:i] = [x]	Insert x into the list before the index	[1,2,3].insert(1, 4) --> [1,4,2,3]
l.pop({index})	Removes the nth (default last) item	[1,2,3,4].pop(0) --> [2,3,4], 1 [1,2,3,4].pop() --> [1,2,3], 4
l.remove(x)	Removes the item from the list	[1,2,3,4].remove(3) --> [1,2,4]
l.reverse()	Reverses the list (in-place)	[1,2,3].reverse() --> [3,2,1]
l.sort({cmp})	Sorts the list (in-place); The cmp function is used to sort the items. The cmp function takes two argument and returns <0, 0, >0	[1,4,3,2].sort() --> [1,2,3,4]

Appendix D: Map methods

Maps support several useful methods.

Method	Comment(s)
m.clear()	Empty the map
m.copy()	Make a shallow copy of the map
m.has_key(k) -- or -- k in m	Test to see if a key is present
m.items()	Get a list of the key/value tuples
m.keys()	Get a list of the keys
m.values()	Get a list of the values (may have duplicates)
m1.update(m2)	add all the items in m2 to m1
m.get(k{, default}) m.setdefault(k, default)	Get the value of k, return default/KeyError if missing; same as get, but set a persistent default value

m.popitem()	Get and remove some item, used during iteration over the map. Example:
	`m = {1:1, 2:2, 3:3}`
	`while len(m) > 0:`
	`  i = m.popitem()`
	`  print i`

Appendix E: Built-in functions

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 -- func(*pargs {, **kargs})	Execute the function with the supplied positional arguments and optional keyword arguments	apply(lambda x, y: x * y, (10, 20)) --> 200
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, name, value)	Creates/sets the value of the object's attribute	setattr(myObj, "size", 10)
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')]

See the Python Library Reference (Resources) for more details.

Appendix F: Jython library summary

Jython supports a large number of Python libraries. By using only these libraries it is possible to write Jython programs that will work in any Python environment. Many of these libraries provide similar function to those provided by the Java APIs. Jython also has access to all Java libraries. This means it can do anything a Java program can do but then it is no longer possible to run the program in a Python environment.

Most libraries that are written in Python and do not depend on operating system specific services are supported without change. Many of these libraries are shipped with Jtyhon. Libraries written in C must be converted; many of the core C libraries have been converted and are shipped with Jython.

Jython also has a few unique libraries of its own. These libraries supplement the extensive API libraries provided by Java itself. For more details on these libraries, read the source files (in <jython_install_dir>/Lib/<lib_name>.py) or see the Python Library Reference (Resources).

Some of the more interesting external libraries supplied with Jython include:

Library	Comment (often from the library prolog)
atexit	Allows a programmer to define multiple exit functions to be executed upon normal program termination
base64	Conversions to/from base64 transport encoding as per RFC-1521
BaseHTTPServer	HTTP server base class (abstract)
bdb	Generic Python debugger base class
bisect	Some Bisection algorithms
calendar	Calendar printing functions (in English)
cgi	Support module for CGI (Common Gateway Interface) scripts
CGIHTTPServer	CGI-savvy SimpleHTTPServer
cmd	A generic class to build line-oriented command interpreters
code	Utilities needed to emulate Python's interactive interpreter
codecs	Python Codec Registry, API and helpers (abstract)
colorsys	Conversion functions between RGB and other color systems
ConfigParser	Configuration file parser
Cookie	Cookie is a module for the handling of HTTP cookies as a dictionary
copy	Generic (shallow and deep) copying operations
difflib	Utilities for computing deltas between objects
dircache	Read and cache directory listings
doctest	A framework for running examples in document strings (sort of like JUnit); I recommend unittest below
dumbdbm	A dumb and slow but simple dbm clone
fileinput	Class to quickly write a loop over all standard input files

fnmatch	Filename matching with shell patterns
formatter	Generic output formatting framework (abstract)
fpformat	General floating point formatting functions
ftplib	An FTP client class and some helper functions
getopt	Parser for command line options (UNIX style)
glob	Filename globbing (a list of paths matching a pathname pattern) utility
gopherlib	Gopher protocol client interface
gzip	Functions that read and write gzipped files
htmlentitydefs	HTML character entity references
httplib	HTTP/1.1 client library
imaplib	IMAP4 client
imghdr	Recognize selected image file formats based on their first few bytes
isql	Provides an interactive environment for database work
linecache	Cache lines from files
mailcap	Mailcap file handling. See RFC 1524
mimetools	Various tools used by MIME-reading or MIME-writing programs
mimetypes	Guess the MIME type of a file
MimeWriter	Generic MIME writer
mimify	Mimification and unmimification of mail messages
multifile	A readline()-style interface to the parts of a multipart message
nntplib	An NNTP client class based on RFC 977: Network News Transfer Protocol
nturl2path	Convert a NT pathname to a file URL and vice versa
pdb	A Python debugger
pickle	Create portable serialized representations of Jython (not Java) objects
pipes	Conversion pipeline templates
poplib	A POP3 client class
posixfile	Extended file operations available in POSIX
pprint	Support to pretty-print lists, tuples, & dictionaries recursively
profile	Class for profiling python code
pstats	Class for printing reports on profiled python code
pyclbr	Parse a Python file and retrieve classes and methods
Queue	A multi-producer, multi-consumer queue
quopri	Conversions to/from quoted-printable transport encoding as per RFC-1521
random	Random variable generators
re	Regular Expression Engine (clone of sre)
repr	Redo the '...' (representation) but with limits on most sizes
rfc822	RFC-822 message manipulation class
sched	A generally useful event scheduler class

sgmllib	A SAX-like parser for SGML (subset as used by HTML), using the derived class as a static DTD (abstract)
shelve	Manage shelves (persistent, dictionary) of pickled objects
shutil	Utility functions for copying files and directory trees
SimpleHTTPServer	A Simple HTTP Server (text HEAD and GET only)
smtplib	SMTP/ESMTP client class that follows RFC-821 (SMTP) and RFC-1869 (ESMTP)
sndhdr	Routines to help recognizing select sound files
socket	Basic socket support
SocketServer	Generic socket server classes
sre	Regular Expression Engine
stat	Constants/functions for interpreting results of os.stat() and os.lstat()
string	Common string manipulations; a (very useful) collection of string operations. The string type also supports most of these functions as methods.
StringIO	File-like object that reads from or writes to a string buffer
telnetlib	TELNET client class based on RFC-854
tempfile	Temporary files and filenames
threading	New threading module, emulating a subset of the Java platform's threading model
tokenize	Tokenization help for Python programs
traceback	Extract, format and print information about Python stack traces
unittest	Python unit testing framework, based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework
urllib	Open an arbitrary URL
urlparse	Parse (absolute and relative) URLs
user	Hook to allow user-specified customization code to run at start-up
UserDict	A more or less complete user-defined wrapper around dictionary objects
UserList	A more or less complete user-defined wrapper around list objects
UserString	A user-defined wrapper around string objects
whrandom	Wichman-Hill random number generator
xmllib	A SA-like parser for XML, using the derived class as static DTD (abstract)
zipfile	Read and write ZIP files
__future__	Used to access features from future versions that are available (potentially in less than finished form) today

Note: I do not claim the above library modules work or are error free on Jython, especially when you are not running on a UNIX system. Try them interactively before you decide to code to them.

Appendix G: Jython types summary

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 Dynamic type testing). 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).

Appendix H: Format codes

The format operator (see Formatting strings and values 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). Use of case in format characters (for example, *X* vs *x* causes the symbol to show in matching case.

Downloads

Description	Name	Size
Sample code	j-jython1.zip	417KB

Resources

Learn

- Take the second part of this tutorial "Introduction to Jython, Part 2: Programming essentials" (*developerWorks*, April 2004).
- Learn more about adding to or updating the Jython registry file.
- 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 the Python Library Reference, Python docs, and Python tutorials on the Python home page.
- Try your hand at using Jython to build a read-eval-print-loop, with Eric Allen's "Repls provide interactive evaluation" (*developerWorks*, March 2002).
- You'll find articles about every aspect of Java programming in the developerWorks Java technology zone.
- Also see the Java technology zone tutorials page for a complete listing of free Java-focused tutorials from developerWorks.

Get products and technologies

- Visit the Jython home page to download Jython.

Discuss

- Participate in the discussion forum for this content.

About the author

Barry Feigenbaum

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.

Acknowledgements

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