Extending Jython

with SIM, SPARQL and SQL
Outline of topics

- Interesting features of Python and Jython
- Relational and semantic data models and query languages, triple stores, RDF
- Extending the Jython grammar, abstract syntax, code compiler, runtime behavior
- Project suggestions and opportunities
Why is Python lovely?

- High level, multi-paradigmed: object oriented, imperative, functional, the best of many worlds
- Dynamic-duck-strong typing, automatic memory management, garbage collection
- Emphasizes code readability, clear syntax
- Off-side rule block delimiting instead of braces
- Scripting, prototyping, scientific computing, data visualization, web frameworks, Google, bittorrent
- CPython is the mainstream implementation
Isn’t Jython reinventing the wheel?

- Python does not compile into universal binaries which run on every platform
- Jython dynamically compiles Python into Java bytecode which runs directly on a JVM
- Jython programs can interact with other Java packages and libraries
- Sacrifice a little speed for portability
- Extend language with Java tools instead of C
J/Python features and caveats

- Compile or Interpret
- Off-side rule
- Duck Typing
- Multiple return values
- Argument passing
- List comprehension
- Functional programming
- Decorators
Interactive interpreter

$ ./dist/bin/jython
Jython 2.5.1+ (unknown:exported, Feb 24 2010, 14:11:17)
[Java HotSpot(TM) Client VM (Apple Inc.)] on java1.5.0_22
Type "help", "copyright", "credits" or "license" for more information.

>>> from math import *

>>> 2**32/1024**3 # how many gigabytes in 32 bit address space?
4L

>>> 2**64/1024**3 # how many gigabytes in 64 bit address space?
17179869184L

>>> 2**1024 # try doing this on a TI-89 without overflow...
179769313486231590772930519078902473361797697894230657273430081157732675805500963132
7084773224075360211201113879871393357658789768814416622492847430639474124377767893424
865485276302219601246094119453082952085005768838150682342642881473913110540827237163
3505106845862982399472459847971630483535632962422424137216L

>>> log(exp(2)) == exp(log(2)) # are nat log and e^x inverse functions?
True

>>> [(x, log(x)) for x in xrange(1, 11)] # log of first ten positive integers
[(1, 0.0), (2, 0.6931471805599453), (3, 1.0986122888681096), (4, 1.3862943611198906), (5, 1.6094379124341003), (6, 1.791759469228055), (7, 1.9459101490553132), (8, 2.0794415416798357), (9, 2.1972245773362196), (10, 2.302585029914446)]
Offside-rule block delimiting

- Indentation using spaces or tabs at the very left of statements is used to delimit blocks instead of curly braces.

- The exact amount of indentation does not matter, instead the relative indentation of nested blocks matters.

- Bruce Eckel: Because blocks are denoted by indentation in Python, indentation is uniform in Python programs. And indentation is meaningful to us as readers. So because we have consistent code formatting, I can read somebody else's code and I'm not constantly tripping over, "Oh, I see. They're putting their curly braces here or there." I don't have to think about that.
Offside-rule implementation

- The tokenizer uses a stack to store indentation levels.
- Whenever a nested block begins, the new indentation level is pushed on the stack, and an <INDENT> token is passed to the parser, though there can never be more than one <INDENT> token in a row.
- When a line is encountered with a smaller indentation level, values are popped from the stack until a value is on top which is equal to the new indentation level (if none is found, a syntax error occurs).
- For each value popped, a <DEDENT> token is generated, and there can be multiple <DEDENT> tokens in a row.
- At the end of the source code, <DEDENT> tokens are generated for each indentation level left on the stack.
Offside-rule example

Sample code:

```python
if foo:
    if bar:
        x = 1
else:
    x = 0
```

Token stream and indent stack:

```
<if> <foo> <::> [0]
<INDENT> <if> <bar> <::> [0, 4]
<INDENT> <x> <=> <1> [0, 4, 8]
<DEDENT> <DEDENT> <ELSE> <::> [0]
<INDENT> <x> <=> <0> [0, 2]
<DEDENT> [0]
```
Duck typing

```python
1 def function(x, y, z):
2     return (x+y)*z
3
4 >>> print function(1, 2, 3)
5 9
6
7 >>> type(function(1, 2, 3))
8 <type 'int'>

9 >>> print function("one", "two", 3)
10 onetwoonetwoonetwo
11
12 >>> type(function("one", "two", 3))
13 <type 'str'>
```

- An object's current set of methods and properties determines the valid semantics
- Allows polymorphism without inheritance
- The name of the concept refers to the duck test, attributed to James Whitcomb Riley: "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."
Multiple return values

- Python will automatically pack multiple values into a tuple
- You can otherwise pack your own lists, tuples, dicts
Function arguments

```
1 def function(arg0, arg1, *args, **kwargs):
2     print "arg0: %s" % arg0
3     print "arg1: %s" % arg1
4     print "args: ", args
5     print "kwargs: ", kwargs
6
7 >>> function(1, 2, "three", "four", key1="five", key2="six")
8
9 arg0: 1
10 arg1: 2
11 args: ('three', 'four')
12 kwargs: {'key2': 'six', 'key1': 'five'}
```

- Arguments are passed by reference, but immutable types (tuples, ints, strings) behave like pass by value
- Positional arguments precede keyword arguments
- *args receives a tuple (automatically packed)
- **kwargs receives a dictionary (automatically packed)
List comprehensions and generator expressions

- List comprehension is a construct for creating a new list from an existing list
- This is analogous to mathematical set builder notation
- Generator expressions are the lazy evaluation equivalent of list comprehensions
- These are extremely useful, and you should master them!
Anonymous functions

• In Python, lambda functions are a matter of style. You can always define a separate normal function instead.

• Lambdas are appropriate when encapsulating specific, non-reusable code without littering the code base with simple one-line functions.

• List comprehensions and generator expressions are generally used instead of filter() and reduce()

• reduce() has been removed from Python 3.0 standard library
Decorator pattern

- A design pattern for use with any objected oriented programming language
- Uses composition rather than inheritance to extend classes
- Allows new behavior to be added to a particular object at runtime, independently of other instances of the same class
- Java I/O Streams implementation employs the decorator pattern
Function Decorators

```python
def entryExit(f):
    def new_f():
        print "Entering", f.__name__
        f()
        print "Exited", f.__name__
    return new_f

@entryExit
def func1():
    print "-inside func1-"

@entryExit
def func2():
    print "-inside func2-"

>>> func1()
Entering func1
-inside func1-
Exited func1

>>> func2()
Entering func2
-inside func2-
Exited func2
```
Data Models

How well does a model capture the meaning of data and inter-data relationships?
University database

Diagram created by:
Saurab Boyed
Relational data models

- Based on mathematical relations and first-order predicate logic
- By far the most popular model for data management in use today
- Complex relationships are difficult to describe and slow to query over
- Referential integrity issues
- Biased toward the implementation
Semantic data models

- Representation of how data relates to the real world is part of the model (rather than the implementation as per SQL databases)
- Class hierarchies and inheritance similar to objected oriented programming paradigm
- Better for models with hierarchic structure or entity relationships (many-to-many relationships are easy to define and query)
Hacking Antlr and Jython

- /grammar/Python.g
  - sqlquery_stmt, sqlinsert_statement - define new types of small_stmt; decompose SQL statements into respective lists; expressions in where clause stored in separate list
- /src/org/python/antlr/ast/SQLQuery.java
  - SQLGetQueryProcessor - call SQLProcessQuery of current SQLQuery object
  - SQLProcessQuery - retrieve query results from database
- /src/org/python/compiler/CodeCompiler.java
  - visitSQLQuery - push where clause expressions onto runtime stack for evaluation
- /src/org/python/core/Py.java
  - sqlWhereClause - store evaluated where clause expression of current SQLQuery
  - sqlWhereClauseFinal - callback to SQLQuery.SQLGetQueryProcessor sending back evaluated where clause expression
- /src/org/python/antlr/ast/VisitorBase.java, /src/org/python/antlr/ast/VisitorIf.java
  - visitSQLQuery, visitSQLInsert, visitSIMQuery - updates to handle new AST types
Grammar for sqlquery_stmt

```
SELECT | NAME | STAR | COMMA | NAME | CAPSFROM | NAME | COMMA | NAME

WHERE | NAME | ASSIGN | LESS | ... | ALT_NOTEQUAL | expr

CAPSAND | CAPSOR | NAME | ASSIGN | LESS | ... | ALT_NOTEQUAL | expr
```

(boxes with double lines are accepting)
References

- The Python Language Reference
- Quick reference for Python 2.6
- The Jython Wiki
- The Definitive Guide to Jython
- SPARQL Query Language for RDF
- A Semantic Database Management System: SIM
- Web Database (WDB): A Java Semantic Database