Closing the Case for Groovy (and Ruby, and Python)

Dr Russel Winder

Concertant LLP

russel.winder@concertant.com

© 2007 Russel Winder

Aims and Objectives

Convince people that dynamic typing is not a difficulty, its an opportunity.

Convince people that closures are a great tool.

Foment thinking and debate.

Exception handling and dynamic binding are a core part of the structure of this session.

© 2007 Russel Winder

Subliminal (!) Advertising

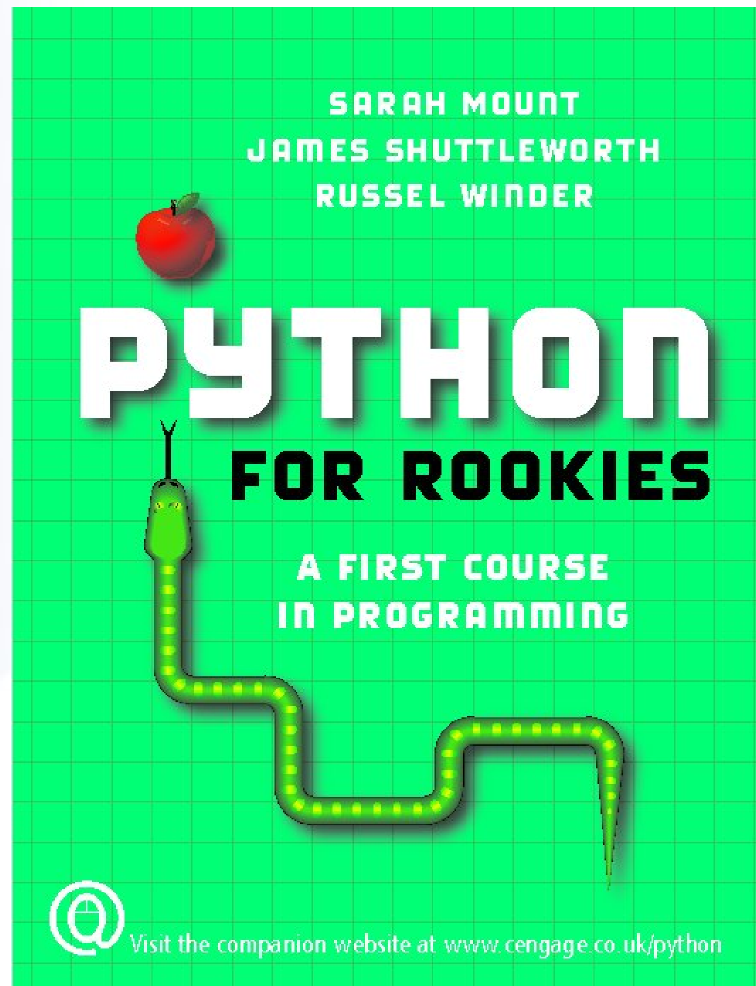
Python for Rookies

Sarah Mount, James Shuttleworth and Russel Winder

Cengage Learning

Formerly Thomson Learning.

Learners of Python need this book.



Draft cover, things may change.
Python is, after all, a dynamic language.

The Strong Typing Imperative

Software engineering dogma demands abstract data types and strong (= static) typing.

Type checking as:

- Pre-execution testing of code.
- Tool for not getting it wrong.

C++, Java, Ada, Eiffel, Scala, Haskell are paradigms of languages meeting the requirements of software engineering doctrine.

Imperative languages not mandated.
Functional languages are acceptable.
Object-orientation has hegemony.

© 2007 Russel Winder

Programming to Interfaces

- Java really popularized *interfaces*, though C++, Eiffel etc. have them.
- Classes are for creating objects, interfaces are for defining the public interfaces. cf. List in Java.
  - List<T> is the interface defining “listness”.
  - ArrayList<T>, LinkedList<T> provide representations of List<T> with different properties.
- Code uses types not classes.
- Correct use of types checked at compile time.

© 2007 Russel Winder

Polymorphism

Generic programming and polymorphic functions increasingly seen as the way forward.

Programming to interfaces supports polymorphism.

In statically typed languages, there has to be an underlying type relationship to achieve polymorphism: *inheritance* or *conformance* required.

Do we agree that programming to interfaces and polymorphism are good?

© 2007 Russel Winder

Inferring the Types

C++, Java, Ada require explicit naming of types in all places – duplication of typing (on the keyboard).

```
final List<T> l = new ArrayList<T> ( ) ;
```

Type inference systems have been around for ages – functional languages.

```
val l = new Array[T] ( )
```

The type structure and checking are the same but the compiler does the work.

Scala and Haskell have this, C++ is getting it (a bit). Why are Java and C# so backward looking?

© 2007 Russel Winder

Strong ≡ Static ?

All the languages mentioned so far as being strongly types are compiled languages:

C++, Java, C#, Scala, Haskell

Are there any dynamic languages that are strongly typed?

Python, Ruby, Groovy

Duck now…

© 2007 Russel Winder

Who Put the Duck in Typing

In*duck*tive reasoning.

If it looks like a duck, walks like a duck, swims like a duck, and quacks like a duck, then it must be a duck.

© 2007 Russel Winder

Not Inferring the Types

Take a simple step:

- Enforce programming to interfaces.
- Do the check at run time instead of compile time.

Are there adherents?

- Smalltalk
- Python
- Ruby
- Groovy

If an object responds to the method, it must be of the right type.

Is this the ultimate expression of polymorphism*?*

© 2007 Russel Winder

The Role of Testing

Testing is critical for programming.

Programming in C++, Java, Haskell, etc. requires test suites using test frameworks:

- Aeryn, TestNG, HUnit.

What is good for statically typed languages is good for dynamically typed languages.

Programming in Smalltalk, Python, Ruby, Groovy, etc. requires test suites using test frameworks:

- sUnit, PyUnit, Test::Unit, Test’N’Groove.

© 2007 Russel Winder

Dynamic Languages and Testing

It is true that dynamic languages rely more on unit testing that static languages do.

Is this a problem?

© 2007 Russel Winder

Having it Both Ways

Groovy is a dynamic programming language, with strong but dynamic typing.

- Groovy is compiled before executed – compiled to JVM bytecodes.
- Groovy works symbiotically with Java uses the Java data types – the object forms by default, can also use the primitives when calling Java code.
- Systems that are part Groovy and part Java are natural and work very well.
- Static where needed, dynamic where wanted.

© 2007 Russel Winder

What's a Closure?

A closure is an (anonymous) function with no free variables.

Lexical scoping to provide bindings to free variables in an anonymous function.

Prerequisite: functions as first-class entities. (In some form or other.)

© 2007 Russel Winder

Who's in and Who's Out

No closures:

- Assembly language
- C
- C++
- Fortran
- Ada
- *Java*
- *Python*

Has closures:

- Scheme
- Smalltalk
- Scala
- Haskell
- Erlang
- Ruby
- *Groovy*
- *C#*

© 2007 Russel Winder

The Fowler Ruby Examples

```
def managers(emps)
  return emps.select {|e| e.isManager}
end
```

```
def highPaid(emps)
  threshold = 150
  return emps.select {|e| e.salary > threshold}
end
```

```
def paidMore(amount)
  return Proc.new {|e| e.salary > amount}
end
```

```
File.open(filename) {|f| doSomethingWithFile(f)}
```

© 2007 Russel Winder

As Lisp – Stefan Roock

```
(defun is-manager (emp) (getf emp :is-manager))

(defun managers (emps)
  (remove-if-not (lambda (emp)
                   (when (is-manager emp) emp))
                 emps))
```

```
(defun high-paid (emps)
  (let ((threshold 150))
    (remove-if-not (lambda (emp)
                     (when (> (getf emp :salary) threshold) emp))
                   emps)))
```

```
(defun paid-more (amount)
  (lambda (emp) (when (> (getf emp :salary) amount) emp)))
```

As Python – Ivan Moore

```
def managers(emps):
    return filter(lambda e: e.isManager, emps)
```

```
def managers(emps):
    return [e for e in emps if e.isManager]
```

```
def highPaid(emps):
    threshold = 150
    return filter(lambda e: e.salary > threshold, emps)
```

```
def highPaid(emps):
    threshold = 150
    return [e for e in emps if e.salary > threshold]
```

```
def paidMore(amount):
    return lambda e: e.salary > amount
```

© 2007 Russel Winder

As Groovy

```
def managers ( emps ) {
  emps.grep { e -> e.isManager }
}
```

```
def highPaid ( emps ) {
  threshold = 150
  emps.grep { e-> e.salary > threshold }
}
```

```
def paidMore ( amount ) {
  { e -> e.salary > amount }
}
```

© 2007 Russel Winder

Java and Closures

Anonymous classes

BGGA proposal for Java 7

© 2007 Russel Winder

Python

Doesn't have closures per se.

Functions are first class.

Closures can be (sort of) mocked up by using higher-order functions.

© 2007 Russel Winder

Groovy

Has things called closures but technically they are not in the classic sense.

Names bound at run time, not lexically scoped.

© 2007 Russel Winder

The Effect on Programming Style

Being able to pass functions as parameters, that can be made into closures, changes control flow.

Remove explicit iteration:

```
for ( i in x ) { println ( i ) }
```

```
x.each { i -> println ( i ) }
```

Make things more declarative, functional even.

© 2007 Russel Winder

Thinking Functionally

- Closures lead to more functional programming ways of expressing things.
- Functional languages have the higher-order functions:
  - map, filter, reduce, etc. in Haskell
  - (and Python, at least for a while!)
- Smalltalk, Ruby, Groovy all support methods on data structures to give the same sort of expression:
  - Groovy: someList.collect { doSomething (it) }

Closing in on Builders

- Closures allow you to create Builders.
- Builders are the technique of choice for creating hierarchically structured systems.
- In Groovy:
  - GUIs – SwingBuilder
  - XML – MarkupBuilder, StreamingMarkupBuilder
  - Ant – AntBuilder

© 2007 Russel Winder

Summary

Dynamic typing is a technique to use when appropriate.

Mixed statically typed and dynamically typed systems have the best of all world.

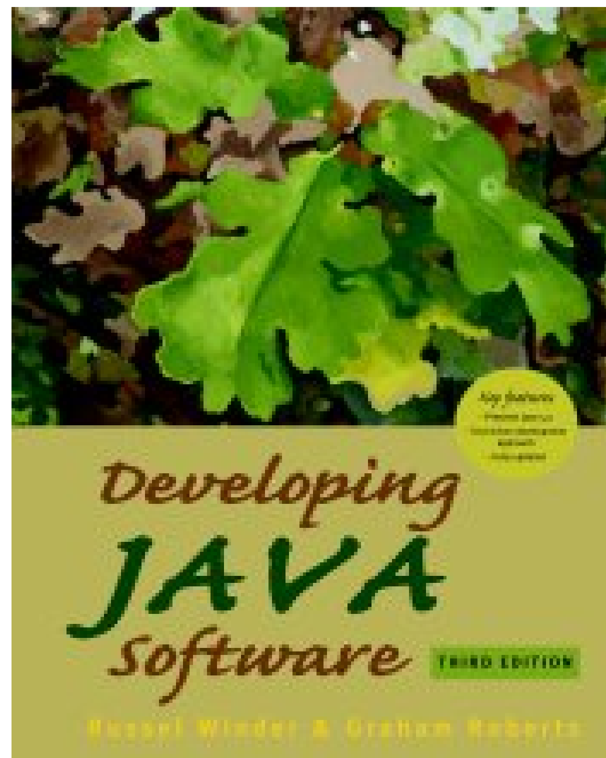
Java + Groovy make a great team.

Closure change the way you program.

And…

© 2007 Russel Winder

Another Advert



Java learners ***need*** *this.*