

# Programming for Chemical and Life Science Informatics

I573 - Week 2  
(Language Overview)

Rajarshi Guha

22<sup>nd</sup> January, 2009

# Outline

- 1 Overview
- 2 Python
- 3 Java

# Programming Practice

## General guides ...

- Much of it is rules of thumb
- Exceptions are always possible
- A variety of style guides are available for different languages
  - ▶ C (also [here](#))
  - ▶ C++
  - ▶ Java
  - ▶ Python
- Useful books include
  - ▶ [The Elements of Java Style](#) by A. Vermeulen et al.
  - ▶ [Elements of C Style](#) by S. Oualline

# Programming Practice

## Documentation

- A tedious thing to do
- Invaluable 6 months later when you've forgotten what the code did
- But bad or unnecessary comments are as bad as no documentation  
`i++ /* increments i */`
- Make use of language specific documentation tools
  - ▶ Javadocs for Java
  - ▶ [Doxygen](#) is very useful for generating docs for many languages (C, C++, etc)

# Programming Practice

## Variables

- Variable naming sounds like a trivial thing
- Can make you program easier to understand, easier to read
- Some languages have developed *traditions*
  - ▶ Java usually uses camel case: each sub word in a variable is capitalized (such as `shortWordCount`)
- Names for loop variables are usually `i`, `j` etc.
- For other purposes one or two letter names are too cryptic
- Use a name that provides some meaning, but should not be too verbose

# Programming Practice

## Functions

- Be careful about non-local changes
- A function should clearly state whether it will make changes to the input arguments or not
- In some cases this is a feature; generally not intended

# Programming Practice

## Functions

- This arises from the use of pointers (or references)
- In C, if you pass a pointer to a variable, then changes to it in the function are visible outside the function

```
int function(int *x, int len) {
    int sum = 0;
    int i = 0;
    for (i = 0; i < len; i++) {
        x[i] = (int) sqrt(x[i]);
        sum += x[i];
    }
    return sum;
}
```

- This is a common procedure in C
- But the fact that the function returns a value and changes an input can be confusing
- Usually the return value is a status indicator

# Programming Practice

## Functions

- But this can become a trap for languages that claim not to use pointers (so preventing memory errors)
- In Python, changes made to an integer in a function are local
- Changes made to list elements are non-local, since the list is passed by value, but the contents are references
- Java has the same behavior

# Programming Practice

## Functions

- In general, argument passing can be of two types
  - ▶ Pass by value (C, Python, Java)
  - ▶ Pass by reference (Fortran)
  - ▶ Pass by reference and value (C++)
- Very easy to trip up, unless you're clear about the distinctions
- Documentation is important
- Language features can also help (`final` in Java, `const` in C++)

# Programming Practice

## Testing

- This cannot be said enough
- Use an appropriate testing framework
- Saves your time and efforts down the road
- Try and develop the test right after you've written the code

# Programming Practice

## Testing

- Make sure that the test data and expected results are correct
- Try and have an automated way to run tests
  - ▶ Test suites in JUnit are handy for this
- Most IDE's will have good support for testing, make use of those
- If you make modifications to your code, make sure to run the tests

# Programming Practice

## Memory issues

- This is not a big problem for programs in Python, Java, Ruby etc
- Mainly relevant for C and C++ programs
- Simple rule
  - ▶ If you allocate memory you *must* deallocate it somewhere
- Note that there are different ways of allocation
  - ▶ Heap - `int *x = (int*) malloc(10 * sizeof(int));`
  - ▶ Stack - `int[10] x;`
- Heap allocations must be deallocated
- Stack allocations can be handled automatically by the compiler

# Choice of Language

- Can start a religious war
- In the end, depends on what problem you're solving
- Traditionally Fortran is the choice for scientific computing
  - ▶ Many math libraries were originally written in Fortran
  - ▶ Much well established code is in Fortran
  - ▶ F95 is a much improved version of the language
- Nowadays, C and C++ compete well with Fortran in terms of performance
- Though Java can be slower than C and C++, it suffices for many applications

# Choice of Language

- In cheminformatics, most problems do not need blazing performance
  - ▶ Doing something in 10 minutes versus 5 minutes is probably not an issue
- Certain application areas definitely do need high performance
  - ▶ Molecular dynamics
  - ▶ SMILES parsing
- For many problems, ease of use is more relevant than performance
- Hence languages and libraries that support rapid development are very useful

# Python

- Easy to learn scripting language
- TOGWTDI as opposed to TMTOWTDI
- Object oriented in nature
- Dynamically typed
- Whitespace irritates some people, but is not really an issue
- Easy to integrate C/C++ code

# Python

- [SciPy](#) is a collection of Python modules that provides a lot of math functions
- [NumPy](#) allows for very efficient multidimensional arrays
- [matplotlib](#) is an excellent plotting library
- Numerous scientific programs written in Python
  - ▶ [PyMol](#) and [Chimera](#)
  - ▶ [mmLib](#) and [MMTK](#) are Python toolkits for molecular modeling
  - ▶ [BioPython](#) toolkit for bioinformatics
- [Software Carpentry](#) is an excellent Python tutorial oriented towards scientists
- [Dive into Python](#) is a good general tutorial
- Python documentation is an excellent resource

# Python

- Interpreted Python is not particularly fast
- But ease of programming improves overall efficiency
- Excellent for prototyping
- Ease of C/C++ integration allows you to gain performance by using C code, but have the ease of use of Python
  - ▶ See [SWIG](#)
  - ▶ See a [presentation](#) by Andrew Dalke on the mechanics of integrating C and Python

# Java

- Traditionally not considered to be high performance
- Not very true these days - it's possible to write fast Java programs
- Portability is a great benefit, very easy to make Java code into web services
- Downside is that there aren't many scientific or math libraries for Java
  - ▶ [Colt](#) well known for Java HPC
  - ▶ [Linear Algebra](#) (specifically LINPACK and BLAS)
  - ▶ [JAMA](#) for matrix operations. Not very efficient or optimized
  - ▶ [JGraphT](#) is a library for graph theoretical algorithms
  - ▶ CDK, JChem, OEChem provide cheminformatics support

# Java

- But Java can be quite slow if poorly or naïvely written
- Don't use synchronized classes if your program is not concurrent
- If you can't improve the performance of some core functionality you can use JNI to interface with a C library
  - ▶ Lose the portability of your Java code
- Budimlic, Z. et al, “[Improving Compilation of Java Scientific Applications](#)”, *Intl. J. High Perf. Comput.*, **2007**, 21, 251–265

# Outline

- 1 Overview
- 2 Python
- 3 Java

# The Python Language

- Completely cross-platform, except for GUI's
- Interpreted, can automatically generate byte-compiled code
- Python uses indentation to indicate code blocks, not braces
- Object oriented nature has some limitations (defects?)
- Good to become familiar with functional paradigms

# Data Types

- Primitive types - int, float, string, boolean, complex
- Complex types - list, tuple, dictionary, set
- Dynamically typed - no need to specify types beforehand
  - ▶ Makes for very easy programming
  - ▶ Can be problematic in large code bases
- Without documentation or explicit type checking, what should you call the function with?

```
def myfunction(x):  
    if len(x) < 0:  
        raise Exception("Invalid input")  
    else: return x[1:]
```

# Data Types

- Strongly typed - once an object is created it's type does not change
- Implies that automatic conversion does not occur

```
>>> a = 'Hello'
```

```
>>> b = 2
```

```
>>> a+b
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

# Data Types

- Remember that variables are just *labels* to a specific object
- See [here](#) for basic overview of data types and other constructs
- Labels can be reassigned to different types of objects

```
>>> a = 'Hello'  
>>> a = 2
```

# Looping

- General idea is to loop over a list using the “for ... in ...” idiom
- Unpacking allows you to elegantly loop over multi-component lists

```
alist = [1,2,3,4]
```

```
for elem in alist:  
    print elem
```

```
for i in range(0, 4):  
    print alist[i]
```

```
blist = [ (1, 'Hello'), (23, 'World'), (1729, 'Bye') ]
```

```
for number, name in blist:  
    print number, ' matches ', name
```

# Looping - List Comprehensions

- The Pythonic way to process list elements
- Lets you write cool, one-liners

```
# square the elements of a list
```

```
x = [1, 2, 3, 4, 5]
```

```
y = [z*z for z in x]
```

```
# create a list of 0's of length n
```

```
zeros = [0 for x in range(0,n)]
```

```
# create a matrix of 0's, n rows, m cols
```

```
zmat = [ [0 for c in range(0,m)] for r in range(0,n)]
```

## Looping - List Comprehensions

- You can return whatever you like for each list element
- You can process multiple lists simultaneously by using `zip`
- Can work on a subset of elements using an `if` clause

```
>>> a = [1,2,3,4]
>>> b = [5,6,7,8]

>>> zip(a,b)
[(1, 5), (2, 6), (3, 7), (4, 8)]

>>> [x+y for x,y in zip(a,b)]
[6, 8, 10, 12]

>>> [x+y for x,y in zip(a,b) if x % 2 == 0]
[8, 12]
```

# Functional Programming

- Can be a complex topic
- For Python programming 3 functions let us avoid explicit loops
  - ▶ `map(func, list)` - apply `func` to each element of a list and return the results as a list. If  $n$  lists are provided `func` should take  $n$  arguments
  - ▶ `filter(func, list)` - returns the elements of `list` for which `func` returns true
  - ▶ `reduce(func, list)` - Applies `func` (which takes 2 args) to the first 2 elements of the list, then applies `func` to the return value and the third element ...
- Depending on the scenario, `map` and `filter` can be replaced by list comprehensions
- Whether you use these functions or not is up to you

# Functions

- Functions are first class objects - you can assign them to variables, make lists etc
- Can return any type of object, unpacking applies to functions as well

```
def myFunction(x,y):  
    return x+2, y / x
```

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

```
a, b = myFunction(2, 8)
```

```
flist = [myFunction, yourFunction]  
for f in flist:  
    print f(5,3)
```

## Functions - Caveat

- Python uses *pass by value* semantics - saves us from pointers
- Be careful of aggregate type (lists etc) - object is passed by value, but it's elements are references!

```
def func1(x):  
    x = x * 3
```

```
def func2(x):  
    x[0] = 3
```

```
>>> a = 2; b = [1,2,3]; c = [5,6,7]
```

```
>>> func1(a); func1(c); func2(b)
```

```
>>> a
```

```
2
```

```
>>> b
```

```
[3, 2, 3]
```

```
>>> c
```

```
[5,6,7]
```

# Classes

- Python provides OO features - encapsulation, inheritance, etc
- There is no concept of private - a caller can access internals of a class
  - ▶ Be polite, and don't
- Can't overload a constructor, can overload everything else
- Very good for implementing specific semantics
  - ▶ If semantics of an object implies list behavior, implement list indexing
  - ▶ If semantics imply key-value pairs, implement dictionary methods

```
p = Protein()  
  
# print first residue  
print p[0]  
  
# loop over all residues  
for residue in p:  
    print residue
```

# Classes

```
class MyClass:  
    def __init__(self, arg1, arg2):  
        self.var1 = arg1  
        self.var2 = arg2  
  
    def getArg1(self):  
        return self.var1  
  
    def __str__(self):  
        return str(self.var1) + ', ' + str(self.var2)
```

```
>>> o1 = MyClass(1,2)
```

```
>>> print o1
```

```
1,2
```

```
>>> print o1.getArg1()
```

```
1
```

# Useful Libraries

- `xml.etree.ElementTree` for XML parsing - *extremely* handy, Pythonic interface to XML documents
- `urllib`, `urllib2` for connecting to the network
- `os`, `os.path`, `shutil` for file and directory handling
- `optparse`, `getopt` for dealing with CLI's
- `csv` for dealing with CSV files
- DB-API provides uniform interface to RDBMS's

# Example - Get Bioassay Targets

```
import xml.etree.ElementTree as ET; from xml.etree.ElementTree import XML
from StringIO import StringIO
import urllib, urllib2, gzip, re
```

```
query = 'HIV'
```

```
params = urllib.urlencode({'db': 'pcassay', 'retmode': 'xml', 'term': query})
```

```
req = urllib2.Request(searchUrl, params)
```

```
root = XML(urllib2.urlopen(req).read())
```

```
ids = [x.text for x in root.findall('IdList/Id')]
```

```
def getProteinID(aid):
```

```
    req = urllib2.Request(fetchUrl % (aid))
```

```
    resp = urllib2.urlopen(req).read()
```

```
    root = XML(gzip.GzipFile('', 'r', 0, StringIO(resp)).read())
```

```
    path = addns('PC-AssaySubmit/PC-AssaySubmit_assay/PC-AssaySubmit_assay_descr/PC-AssayDescription')
```

```
    pid = root.find(path)
```

```
    if pid is not None: return pid.text
```

```
pids = [getProteinID(x) for x in ids]
```

# Outline

- 1 Overview
- 2 Python
- 3 Java

# The Java Language

- Completely cross-platform, including GUI
- Compiled, based on a virtual machine
- Fully object oriented, but a little more restricted than C++
- Absence of functional paradigms
- Huge amount of external support
- Not always good for quick one-off projects
  - ▶ But see [BeanShell](#), a scripting environment for Java
- See [here](#) for tips on improving Java performance

# Data Types

- Primitive types - int, float, double, boolean, String
- Complex types - List, Set, Map
- Typing static and strong - so you have to specify types beforehand
- Possible to bypass this when using aggregate types (i.e. Collections)
- All classes in the JDK are derived from the Object class
- Primitive types have OO counterparts
  - ▶ int → Integer
  - ▶ double → Double
- From JDK 1.5, you can use either one interchangeably (autoboxing)

```
map.put( new Integer(1), "Value" ); /* JDK 1.4 */  
map.put(1, "Value" );             /* JDK 1.5 */
```

# Use Case

- A Molecule class has a field nAtom
- How do we indicate that the field is unset?
- Using an Integer type allows you to clearly indicate that a field is not set
- If you forget to set up the field properly, accessing it will throw a run time exception
  - ▶ Your code should perform the appropriate checks though

```
public class Molecule {  
    int nAtom = 0; // or -1 is better  
}  
  
public class BetterMolecule {  
    Integer nAtom = null;  
}
```

# Autoboxing Caveats

- Simplifies code - blurs distinction between primitive types and their OO counterparts
- Does not eliminate the difference!
- Autoboxing incurs a performance hit
- Don't use autoboxing in performance sensitive code (but best to use a profiler to see whether there is a difference)
- `==` behaves differently
  - ▶ with `Integer` performs reference identity
  - ▶ with `int` performs value equality

# Data Types

- Java has a rich collection of data structures
- The Collections package is very useful
  - ▶ Set
  - ▶ List
  - ▶ Queue
  - ▶ Map
- These are *interfaces* - descriptions of what a data structure that is supposed to be a “List” should act like
- Vector and ArrayList are implementations of the List interface
  - ▶ Use them in identical fashions
  - ▶ API will be the same

```
List myList = new Vector(); // new ArrayList()
myList.add("Hello World");
myList.add(new Double(10));
System.out.println(myList.contains("Goodbye World"));
```

# Parametrized Types

- Collection types allow you to add any type of object
- Bypasses compile time checking. Easy to forget what's in a container
- JDK 1.5 supports parametrized containers - specify the type of the contents
- Compiler can catch errors

```
List oldList = new ArrayList();  
List<String> newList = new ArrayList<String >();  
  
oldList.add("Hello");  
oldList.add(new Double(1.2));  
  
newList.add("World");  
newList.add(new Double(3.4)); // won't compile
```

# Synchronized Classes

- Java has a number of features for concurrent programming
- Lots of things to be aware of when writing such code
- Simultaneous modification of a data structure
  - ▶ How do we deal with two process trying to modify a single data structure?
- Java makes certain classes `synchronized` - it is safe to use these in concurrent code
  - ▶ `Vector` is synchronized, `ArrayList` is not
  - ▶ `Hashtable` is, `HashMap` is not
- But synchronized code is slightly slower than unsynchronized
- So if you're program is not meant to be multi-threaded, avoid synchronized classes

# Java Classes

- Every Java program is a class
- If it is an application should have a `main` method somewhere
- Avoid writing a program as one big class (unless it's very simple)
- Make use of the Java package structure to divide your code into functionally related components
- Very easy when you're using an appropriate IDE

```
public class SimplestPossibleClass {  
    public static void main(String[] args) {  
        System.out.println("Hello World");  
    }  
}
```

# Java Permissions

- Java has the facility to prevent callers accessing fields, methods etc
- JDK [docs](#) describe it nicely
- Allows you to implement good OO design (unlike Python)
- Generally, choose the most restrictive access. This will simplify large code bases and allow compile time checking

# Java OO Design

- Can cover half a semester on this topic
- Highlight some topics in this area - fundamentally, applications of design patterns
- Look at Open Source projects for real-world examples
- All starts with the class - code and data representing a concept and its behavior
- Objects are instances of a specific class
- OO design is all about how we design and organize classes and collections of classes
- Some aspects are obvious and map easily to common sense
- Some aspects are more subtle, sometimes debatable

# Encapsulation

- By default class variables are private, but accessible to all methods of the class
- Provide access via a getter and setter
- If a variable is involved in a loop, good idea to make it local to the method
- Encapsulation allows you to change the internals of the class, without affecting code that uses the class

```
public class ExampleClass {  
    int var1; // private  
    public var2  
  
    int getVar1 () { return var1; }  
  
    void setVar1 (int newValue) { var1 = newValue; }  
}
```

# Polymorphism

- Multiple methods with the same return value, but different arguments
- Allows you to provide simplified interface to some methods
- Polymorphic constructors are allowed

```
public class Depictions {  
    Image getDepiction(IAtomContainer molecule) {  
        return this(molecule, 200, 200, true);  
    }  
    Image getDepiction(IAtomContainer molecule,  
                       int x, int y, boolean showH) {  
        // generate a 2D depiction of width x, height y  
    }  
}
```

# Inheritance

- Add new behavior to a class without modifying the behavior of the original class
- Dictate that classes must exhibit certain behaviors (via abstract class)
- Consider a class representing molecules, behavior includes getting the number of atoms & bonds, adding atoms, bonds etc

```
public class AtomContainer {  
    int getAtomCount() { /* do stuff */ }  
    void addAtom(IAtom atom) { /* do stuff */ }  
    void addBond(IBond bond) { /*do stuff */ }  
    Iterator atoms() { /*do stuff */ }  
}
```

# Inheritance

- An amino acid is a molecule
- But we'd like to easy get the C- and N-terminus atoms
- Make use of the `AtomContainer` class - but add some new methods to support the new behavior

```
public class AminoAcid extends AtomContainer {  
    void setCTerminus(IAtom atom) { /* do stuff */ }  
    void setNTerminus(IAtom atom) { /* do stuff */ }  
    IAtom getCTerminus() { /* do stuff */ }  
    IAtom getNTerminus() { /* do stuff */ }  
}
```

# Abstract Classes

- Useful way to define what a group of classes should do - but defer implementation (in time, to other people etc)
- Can provide some method implementations
- Cannot instantiate an abstract class
- Designed to be subclassed

```
public abstract class MoleculeBase {
    int getTitle() { return this.title; }

    abstract Iterator atoms();
    abstract String getSymbol(); // AA's can be 1 or 3 letter
}

public class AminoAcid extends MoleculeBase {
    // must implement atoms()
    // and getSymbol()
}
```

# Interfaces

- An interface is a contract - any class implementing an interface will provide implementations of methods in the interface
- Equivalent to an abstract class with all abstract methods
- Interfaces can extend other interfaces

```
public interface IAtomContainer {  
    int getAtomCount();  
    void addAtom(IAtom atom);  
    Iterable<IAtom> atoms();  
}
```

```
public class MyClass implements IAtomContainer {  
    // must implement all methods in IAtomContainer  
}
```

# Abstract Class or Interface?

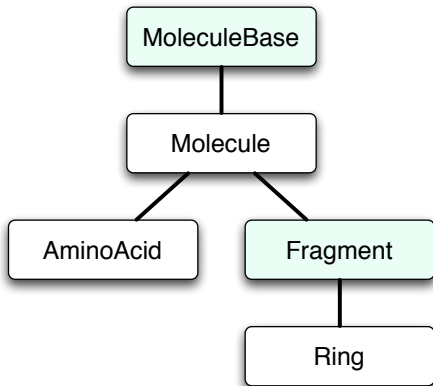
## Differences

- Abstract class can contain methods, non-static fields
- An abstract class that has *only* abstract methods is an interface - should be rewritten as such

## Which One?

- Java doesn't support traditional multiple inheritance
- A class can extend one class, but implement multiple interfaces
- Interfaces can simplify code and class hierarchies
- Use an abstract class when you want a pre-defined hierarchy
- Interfaces are useful when design might change frequently

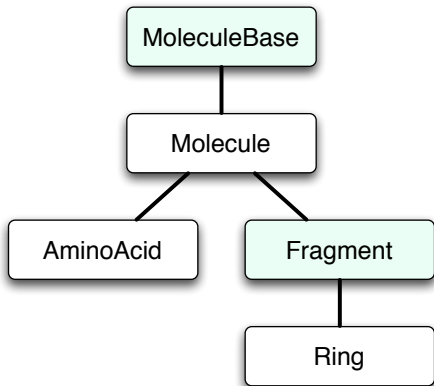
# Abstract Class Example



- A method capable of handling fragments
- Cannot coerce AminoAcid into Fragment
- Cannot coerce Molecule into Fragment
- Possibly have to redesign hierarchy

```
float getMW(Fragment mol) {  
    // do stuff  
}
```

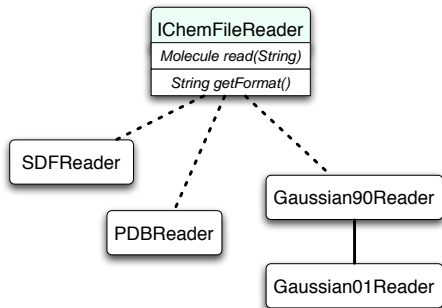
# Abstract Class Example



- A method capable of handling any type of molecule
- The method will not know about any specializations in the subclasses
- Cannot deal with an object that is not a molecule, but could be characterized using molecular weight

```
float getMW(MoleculeBase mol) {  
    // do stuff  
}
```

# Interface Example



- A method reading any type of chemical file
- Implementing classes may or may not be related to each other
- But they have a common behavior - reading a file
- Design is now more flexible

```
Molecule loadFile(
    IChemReader reader,
    String f) {
    return reader.read(f);
}
```

# Handy Libraries

- [JUnit](#) - invaluable for unit testing
- [CLI](#) - for dealing with command line args
- [XOM](#) - very fast, easy to use XML parsing library
- [log4j](#) - handy logging framework
- Getting comfortable with [Ant](#) is very handy
- CDK and JChem are Java cheminformatics libraries
- [BioJava](#) deals with bioinformatics