Introduction to Computer Science and Programming for Astronomers

Lecture 1.

István Szapudi

Institute for Astronomy
University of Hawaii

January 15, 2014
Outline

1. Introduction
   - Course Syllabus

2. A Practical Guide To Coding Principles
   - Unstructured Programming
   - Programming Paradigms
   - Structured Programming

3. Demonstration of The Top-Down Approach
Computational Astronomy– Astro 735

- Wed 14:00 in C-221
- phone: 956-6196
- email: szapudi@ifa.hawaii.edu
- www: www.ifa.hawaii.edu/users/szapudi/astro735
- office hours by appointment
Principal Goals

- to internalize a practical programming philosophy.
- to learn about the language and paradigms of contemporary computer science.
- to learn about basic data structures and algorithms.
- As a side effect, we will learn about **python** and object oriented programming.

735 Computational Astronomy
How to Become a Good Programmer?

- Programming is a skill (try to learn playing violin from a lecture or a book) \( \rightarrow \) needs disciplined practicing.
- Homeworks are important in this class, please take them seriously.
- We will use `python (2.x)` language.
- No textbook (but there are infinite resources on the internet!)
- You should try to look at well written programs.
- Have fun!
You can do 50% of homeworks or a project

- Homework (due each week at beginning of the class, late homeworks not accepted)
- Project (I can give extra credit for exceptionally good projects at my discretion).
- If there are good projects, we can do Final Presentations
Homework Guidelines

Please follow carefully these conventions, since you will be often submitting several files. These rules are designed (hopefully) to avoid confusion.

- Each homework will consist of a file `name-hwn.tar.gz`, where `name` is your unique ifa user name, and `n` is the homework number.
- Each tar-file will open into a directory `./name/hwn/`
- The directory will contain all the files corresponding to the given homework. The accepted file formats are `.pdf`, `.txt` for answering questions, documenting programs, and `.py` for python programs. In addition the directory should include all extra files which your programs might need, i.e. it should be self contained.
Homework Guidelines Cont’d

- Put all your textual answers to questions into one file `ANSWERS.ext` where `ext = .pdf` or `.txt` (clearly, if you submit figures, it should be `.pdf`).

- The naming conventions for submitted python programs should be explained clearly in the `ANSWERS` file.

- Each python program should self-test at command line, e.g., `./qn.py`.

- Homeworks will be emailed to `szapudi@ifa.hawaii.edu`.

- Non-conforming homeworks/programs will be rejected.
Projects

- Need to be approved by the instructor in the first month.
- Preferably it will be something useful for your own research.
- If you have no idea I will give you something.
- You will have time to submit project by the end of the semester.
- Testing, Test-data and Documentation! (Whatever it does I need to be able to run it).
- Final Presentation (TBD).
Language is not important!

Pseudo-code: a language to describe algorithms.

We use python as pseudo-code.

The advantage is that you can run the algorithms almost immediately.

The disadvantage is that you have to be conscious between the difference between implementation and general algorithmic principles.

Note: Since I use python as if pseudo-code, I don’t necessarily use the most efficient implementation.

The main concern in this class is clarity.
python vs. other languages

- Language is important in practice!
- A dynamically typed vs. static
- Interpreted vs. compiled
- Object oriented support (although not necessary).
- Very short development cycles.
- Clear (and beautiful) syntax.
- Extensive libraries (I recommend numpy, scipy, pyfits, and MatPlotLib)
- Open Source.

A minimalist toolkit could consist of python, latex, and emacs (could add C(++), Mathematica, SQL, Office, IDL, Matlab, Fortran, Java, cgi, php, perl, iraf etc.)
Questions?
A set of rules and practices that proved useful for creating programs that are readable and **work**.

My distillation is by no means exhaustive and reflects my personal preferences as well.

We will use the word “structured programming” in loose sense to describe our approach.

Part of the principles are almost aesthetic in nature, yet immensely useful.
The Zen of Programming

What is NOT Structured Programming

10    DIM X(100), Y(100), Z(100)
20    INPUT N, S, S0
30    S = 0
40    FOR I=1 TO N
50    INPUT X(I), Y(I), Z(I)
60    NEXT I
70    FOR I=1 TO N
80    IF X(I)=0 THEN 200
90    X(I) = -X(I)
100   A=0
110   FOR J=1 TO N
120   IF J = I OR X(J) <=0 THEN 140
The Zen of Programming (Cont’d)
What is NOT Structured Programming

130   IF  SQR ((-X(I) - X(J)) ** 2 + (-Y(I) - Y(J)) ** 2
       + (-Z(I) - Z(J)) ** 2) <= S1 THEN X(J) = -X(J) : A = A + 1
140   NEXT j
150   X(K) = 0 : K = K + 1 :
160   IF  K > N THEN 170
170   IF  X(K) >= 0 THEN K = K + 1 : GOTO 160 ELSE 110
180   IF  A > S0 THEN S = S + 1
190   NEXT I
200   PRINT S
210   STOP

Can you understand this? I can’t...
Tell-Tale Signs of the Unstructured Approach

“Natural Programming”

- Obscure form and content (variable names, no comments, no indentation).
- No program design: trial and error, lots of backtracking.
- Once it’s done it’s difficult to debug.
- Difficult to correct bugs: bug fixes create new bugs, a never-ending process
- Difficult to use, impossible for anybody else then the author.
- Cannot be modified, impossible to be reused.
- It’s only matter of time before its own author won’t be able to use it.
- This makes programming a pain!
There are many competing (and fun!) programming paradigms in computer science. E.g.,

- Structured programming vs Unstructured programming
- Procedural vs. Functional (Imperative vs Declarative)
- Object oriented programming
- Constraint programming (Logical programming)
- Even more! (Component based software development, Aspect oriented, Post object, Relational Symbolic programming)

We will use the word “structured programming” (a 70’s concept) in a loose sense to describe a collection of practical methods that work. (One definition: no goto’s).
Main Principle: Divida et Impera
Top Down Approach

This is the most important programming principle (or philosophy?)

- Divide the problem into a few subproblems with well-defined boundary.
- Then recursively apply the same principle on each subproblem (like a pyramid)
- Initially do not try to figure out how just what needs to be done
- The top is the original problem, the bottom level is coding small sub-problems.

Sounds easy? Actually it is an art to apply it skillfully in practice. The temptation to stray away from this simple idea is strong, but you will pay dearly if you do.
Basic (Strategic) Principles

These points help you to apply the above general thought in practice:

- Parallel refinement of levels.
- Delegate decisions to the appropriate levels.
- Backtracking to higher level if you got into a dead end.
- Open design: try to solve a class of problems instead of the original specific problem (keep generalizations in mind).
- Explicit decisions (i.e. try to avoid implicit assumptions)
- Principle of Isolation: different parts should interact only clear, specific and pre-determined way at the boundary
More (Technological) Principles

These principles are at a lower level than the previous set, but equally useful.

- Clearly separate algorithm/program design from coding.
- It is healthy to use pseudo-code to describe algorithms, but *python* is even more efficient.
- Use succinct names for variables (what they *are*, sometimes useful to append units), and functions (what they *do*).
- Specify the problem you want to solve very clearly
- Testing/Quality Assurance is an important part of coding from the very beginning: test each component as you code. Design *unit tests* with your code.
Guidelines for API’s

The language of programming is cluttered with acronyms: API stands for Application Programming Interface.

- Principle of “politeness”
  - help: should tell what it’s doing, what data it needs.
  - verbose running option (program which starts up and churning away without any sign of life is scary)

- Foolproof (robust) design: if there is any way to crash a program, the user will figure it out.

- Readable programs: others (and yourself later) should be able to quickly understand and modify your code.

- Documentation (documentation, documentation!)
Points of User Friendliness

Remember, the user might be yourself in a few months later...

- Write data on the screen in a neat, uncluttered fashion with possible page-breaks.
- Give convenient options to drive the program via menus, command line options (even GUI’s in some situations).
- Don’t forget units (e.g., $h^{-1} Mpc$) and range of parameters.
- Use sensible default values.
- Highlight the most important data/output.
- Spelling, grammar etc apply
- Succinct error reporting and handling
An Example Problem

Simple Friends of Friends algorithm

Let us define clusters as galaxies in groups of at least 5 members, where each member has at least one neighbor from the same group ("friend") within 1 Mpc. Given a galaxy catalog (a list of x,y,z positions), find the number of clusters.

- The previous unstructured program is an attempt to solve this problem
- This is a relatively simple problem, but complex enough that we can illustrate the top-down approach by it
- Remember again that we use Python as “pseudo-code which can be run”, therefore the simple solution I will propose will not be very “pythonic”. 
Immediate Generalizations

- Instead of the concrete problem we solve a whole class of problems, where the link-length (1Mpc) can be arbitrary
- We keep the threshold (5 galaxies) arbitrary
- We could solve the problem in arbitrary dimensions
- We also keep in mind that we might later ask more questions, such as distribution of clusters, etc.
Data Structure

- The first thing we think about is how to represent the data.
- The input data is simply a list of galaxy positions, call it `pos[0..nTot-1][0..dim-1]` for now. For clarity we use `galaxies`, simply a list of indexes.
- While the output is a simply a number (the number of clusters) we could think that a structure representing galaxy groups would be interesting for a large class of problems.
- Therefore we decide to represent galaxy groups as a list of lists, where the n-th list contains the list of galaxy indexes. Call it `clusters` for now.
- Note that this representation is generic (virtually all computer languages have lists). The basic idea would work in any language.
Since we are using python, a language with object oriented support, we are tempted to create a class for this problem.

- A class is a collection of data, and functions which operate on the data (methods). Its definition represents a template.
- An object is an instance of this template.
- Classes are the hallmark of “Object Oriented” design.
- This is often convenient, and one can achieve good isolation between parts by using classes.
- Critics note that the world cannot be represented with classes (e.g., an object belonging to the “Child” class cannot belong also to the “Parent” class).
A module is a file containing classes and functions which other modules or programs can use.

```python
#!/usr/bin/python

""
ff.py: simple friends of friends algorithm
... short description...

Mon Jan 10 21:25:03 HST 2005

(c) Istvan Szapudi, Institute for Astronomy
"""
The First Class

We start by defining the main class for our problem.

class FriendsOfFriends:
    ""
    from a list of object (galaxy) coordinates in n-dimension (n=1,2,3) create the list of clusters defined by friends of friends
    """
Initializing function
__init__ will be called for each instance

In the first function we put what we always want to be done when an instance is created.

```python
def __init__(self, fname, length):
    
    """
    file fname contains ascii positions
    x [y z]
    
    with comments allowed starting with
    #....comment
    
    input is stored in pos, a list of
    [[x,y,z], ...]
    ```
So far we have written comments only! These document the thought process at the top level as well as used for automatic generation of documentation.

...  
A f.o.f. cluster is defined s.t. each member has another galaxy within distance length. creates a list of the clusters in self.clusters
[[galaxy1,galaxy2,...,galaxyN],.....]

""
Finally, some programming!

We decided to divide the problem of creating the list of clusters into two subproblems: reading the data, and actually creating the list.

```python
def __init__(self, fname, length):
    ...  
    self.pos = self.readColumns(fname)
    self.mkClusters()
```
The `pass` statement means the program is doing nothing. Initially, when you are still only at the program design phase, you can create syntactically correct programs by constructing functions which do nothing, and as you zoom into the problem, you can later fill in the details.

```python
def mkClusters(self):
    """
    create the list of clusters using the friends of friends def.
    """
    pass
```
Reading the data
This is totally Python specific

```python
def readColumns(self, infile):
    """
    read list of x1 [x2 ...] from an ascii file
    with # allowed for comments
    """
    pos = []
    for line in file(infile):
        if line[0] == '#':
            continue
        row = [float(val) for val in line.split()]
        pos.append(row)
    return pos
```
A test function

You can define functions outside of the class with the sole purpose of testing your code.

```python
def test():
    """
    unit test for ff.py
    """
    ff=FriendsOfFriends('ff.dat',0.5)
    print ff.pos
    print ff.clusters
    print ff.numberofClusters(2)
```
Then at the very end of your code you can include something like

```python
if __name__=='__main__':
    test()
```

Then your python module will run from command line. (Initially you can put empty functions for parts which are not worked out yet as shown before). Something along these lines is called test driven development. Continuous testing is essential for dynamically typed environments.
At this point we have a syntactically correct code, which reads the data. We can start thinking about the actual problem.

- The first thing comes to mind is that we could find for each galaxy which group it belongs.
- Then simply adding up the groups for each galaxy would constitute our final group.
Chopping Up the Problem.

This thinking leads us to write something like this:

```python
def mkClusters(self):
    '''
    create the list of fof clusters
    '''
    self.clusters = []
    # find clusters for each gal in pos
    for gal in self.galaxies:
        self.checkOne(gal)
```

However, a little thinking reveals that this in itself would multiply count groups...
Equivalence Relations

Let me remind you of equivalence relations:

- Reflexivity \((a \equiv a)\)
- Symmetry (if \(a \equiv b\) then \(b \equiv a\))
- Associativity (from \(a \equiv b\) and \(b \equiv c\) follows \(a \equiv c\))

You can convince yourself that belonging to a friends-of-friends cluster is an equivalence relation. This means that a galaxy can belong to one and only one cluster.
Once a galaxy is assigned to a cluster, we don’t need it further. This leads us to introduce an auxiliary variable, which keeps track of which galaxies have already been assigned to groups.

```python
def mkClusters(self):
    self.checked = self.nTot*[0]
    self.clusters = []
    for gal in self.galaxies:
        if(self.checked[gal] == 0):
            self.checkOne(gal)
```
def checkOne(self, gal):
    """
    check gal for cluster membership
    return the cluster defined by this galaxy
    """
    newCluster = []
    newCluster.append(gal)
    self.checked[gal] = 1
    nCluster = 1
The trick is that we not only have to include friends of our galaxy, the all the friends of its friends as well.

```python
def checkOne(self, gal):
    ...
    k = 0
    while k < nCluster:
        nCluster += self.addFriends(newCluster, k)
        k = k+1

    self.clusters.append(newCluster)
```
addFriends
Initialize

We now can concentrate on adding friends of just one galaxy to the cluster.

def addFriends(self, cluster, k):
    """
    add all galaxies to cluster which are closer than self.length (i.e. Friends) of the k-th cluster member
    return the total number of objects added
    """
    # this will store the number objects added
    n = 0
addFriends Cont’d
We are almost done!

checked will make sure that we cannot add the same galaxy twice

... for gal in self.galaxies:
    if (self.checked[gal] == 0):
        if self.dist(gal, cluster[k]) <= self.length:
            cluster.append(gal)
            n += 1
            self.checked[gal] = 1

return n

Remember we need to return the number of objects added to the cluster!
def calcDist(self, gal1, gal2):
    
    return the Euclidean distance between galaxies gal1 and gal2

    from math import sqrt as sqrt

    d = 0.0;
    for i in range(self.dim):
        d += (self.pos[gal1][i]-self.pos[gal2][i])**2
    return sqrt(d)
Review Of The Core Algorithm

In python-like pseudo-code

```python
clusters = []
for gal in galaxies:
    if not_member_of(clusters, gal):
        newCluster = []
        newCluster.append(gal)
        for gall in newCluster # (includes gal):
            addFriends(newCluster, gall)
        cluster.append(newCluster)
```
Our Original Question
now it’s easy...

```python
def numberOfClusters(self, n0=0):
    """
    returns the number of clusters greater or equal to n0
    """
    nt = len(self.clusters)
    nc = 0
    for i in range(nt):
        if len(self.clusters[i]) >= n0:
            nc += 1
    return nc
```
We are done with the class

- We **really** chopped up the problem into small, manageable parts
- Python **really works as pseudo-code**: you can immediately test your ideas!
- Some of the functions now could be rewritten in more elegant pythonic form (not our concern at the moment).
- If it’s too slow, we can always rewrite parts in C, and since it is modular, we can change part of the algorithm easily!
- Next time we will look at possible main programs based on this class.
We have looked at a set of practical guidelines for program design and development.

The most important principle is a hierarchical *divide and conquer* or top down approach.

We have looked at a simple example of how to put the basic principles into practice.
Homework # 1
The first homework is warm up...

- Reading: read chapters 1-5 of the python tutorial at http://docs.python.org/2/tutorial/.
- Download from the class homepage the code ff.py. Read it, understand it, and find at least one bug in it.
- Write a program, which uses this corrected class, and able to answer our original question (hint: you will start with import ff to import the class written. It's a 3-4 line program comments excluded)
- Criticize ff.py and try to find at least three instances where it does not fully conform to the principles we outlined.
- How fast is this program? Would it be fast enough to analyze an N-body simulation? If not, do you have any ideas to remedy the situation?