Introduction to Computer Science and Programming for Astronomers

Lecture 2.

István Szapudi

Institute for Astronomy
University of Hawaii

January 23, 2014
Outline

1. Reminder
2. Testing and Debugging
3. Some Concepts of Object Oriented Design
4. Selected Data Structures
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.
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)
```
Main Program: nclusters.py
Based on the class written previously

#!/usr/bin/python2.3

import ff
import sys

print "----------------------------------------
print "usage: nclusters.py infile length nmin"
print "----------------------------------------

...
print ""
print "input:"
print "    infile:"
print "    contains positions of objects"
print "    x [y z]"
Main Program: `nclusters.py` Cont’d

The actual program is only 3 lines:

You could do this interactively!

```python
myClass = ff.FriendsOfFriends(sys.argv[1],
    float(sys.argv[2]))
print "total number of galaxies: ",
    myClass.nTot
print "the number of clusters found: ",
    myClass.numberOfClusters(int(sys.argv[3]))
```
Scaling
Our first example for scaling
Main Goal: To Ascertain Correctness

Levels:
- I: (Initial) Testing: run (artificial) “testcases” to find mistakes
- II: (Validation): (test) run on “real data”
- III: (Certification): have been used for a while.
- Debugging: find and correct any problems found during any of the above processes
- Proof of correctness (for completeness: we will not do)
Initial Points on Tests
Good test cases and testing have the properties below:

- Likely to point out a bug not yet discovered (i.e. it is useless to run a test you know will work).
- Test cases include input and output.
- Repeatability is essential.
- Test with both valid and invalid input (the latter should raise error).
- Examine the results of each test carefully from every possible aspects.
- E.g., figure out both i) why it is not doing what is expected, ii) why it is doing something which is not expected.
- Testing is often best done by somebody else (testing is nasty activity). If you test your own work never assume it is correct.
These trivial methods together with good coding practices eliminate most of the bugs despite that the code is not run yet.

- Careful checking of the code can reveal many bugs. A particularly good method is to explain your code to somebody else.
- Syntactic checking (turn on all the warning options on your compiler)
- Semantic (content) checking. Try to run the algorithm in your head...
Dynamic Testing
Checking the program when it’s running

Two-methods:

- **Black box**: we test the program without any regard to the algorithm. I.e. we just check if it performs a given task.

- **“White box”**: we take into account the inner workings (algorithms, data representation) when we design our tests.

These two methods are somewhat complementary, and can catch different kind of bugs.
Black Box Techniques I.
Finding Equivalence Classes

- Try to find equivalence classes in the “configuration space” of possible inputs, s.t. it is likely to be sufficient to test one element in each class.
- This is an efficient way to span configuration space. Don’t stop until each class is covered.
- E.g. the input is a range of values: e.g., \( 1 \geq i \geq 10 \), the three classes could be \( i < 1 \), \( 1 \geq i \geq 10 \), and \( i > 10 \). For more complicated inputs you can use Descartian product of the conditions.
- In general, we should have at least a valid and invalid input class, and divide the classes until we suspect that each important case is covered.
Black Box Techniques II.
Boundary Analysis

- In general many bugs live in **boundaries**! (e.g. first last element of an array).
- Look at the equivalence classes defined before, and test with elements on the boundary.
- Take into account both input and output classes (if any).
- This way can catch bugs which eluded us previously.
White box techniques

```python
if x > 0 and y > 0:
    print x*y
```

- Cover each instruction once (make sure that in your tests each line of code is executed $x = 1$ and $y = 1$)
- Cover each branch ($x = 1$, $y = 1$, and $x = -1$, $y = -1$)
- Cover each condition (related to previous, but $x = 1$, $y = -1$, and $x = -1$, $y = 1$ would cover each condition, but the branch would never be executed)
Examples: White Box
We construct test data for nclusters.py

(length, nmin, (input coordinates)...)  
- Each line is executed if we have at least one cluster, and a # comment in the file e.g.:
  100, 3, (1,1,1), (1,1,2), (1,2,2)
- Each branch: 10, 2, (1,1,1), (100,1,1), (1,1,2)
Examples: Black Box
Still nclusters.py

Equivalence classes from input parameters: valid, invalid, and boundary

- total number of galaxies ($N_{tot} < 2, N_{tot} = 2, N_{tot} > 2$)
- min. cluster size ($< 2, 2, < N_{tot}, N_{tot}, > N_{tot}$)
- length: less than the min. distance, more than the max distance, in between, and boundary
Further Equivalence Classes

- 1,2,3 dimensions
- Galaxies aligned on a line or plane (2,3D, possible directional dependence)
- No clusters
- One cluster
  - Everything is one cluster (percolation)
  - One cluster and “field galaxies”
- 2 or more clusters, etc...
Unit Tests

- A collection of tests which test most aspects of the program.
- Ideally they are written together with the program.
- They are especially important for dynamically typed languages like Python.
- A good practice for Python is to run unit tests every few minutes (test driven development) to make sure nothing is “broken” because of newly written code, and add a new test for the newly written part when needed.
- There is a unittest class in Python which formally supports unit tests. Check it out if you plan to write something big.
Debugging and Correcting
The dreadful part...

- Should never happen if you do everything I suggested thus far (just kidding).
- Before using a debugger, carefully check the code (static testing)
- Don’t start fixing until you figured out exactly the problem (where and why).
- Bugs spread: if you find one in one place, think where else it could cause a problem
- Root out the cause, not just the symptoms
Debugging and Correcting
More on debugging

- After correcting a bug, retest your program to make sure that no new bugs are introduced.
- The likelihood that a bug is correctly fixed is inversely proportional to the size of the program. Isolation, top down approach and object oriented programming try to address this observation.
- In fact the number and gravity of bugs grows as a power of the size of the program.
- Sometimes a program needs to be redesigned because of bugs.
- Python has a pdb class to help debugging. There are great graphical debugging interfaces, like ddd (not for Python). Check them out if you need them.
Debugging Techniques
Useful principles to guess problems: a bit like solving a puzzle...

- **Induction**: (generalization) we try to organize and interpret test results. E.g. $x = 0, 1, 4, 7$ correct, and $x = -1, -2, -5$ incorrect → We assume that the program breaks for negative numbers.

- **Deduction**: (narrowing down) finding the cause of the error. E.g. $x = 3, 5, 7$ incorrect, $x = 12, 18$ correct. Possible assumptions breaks for $x < 10$, odd $x$, prime $x$. We create new tests to narrow down the possibilities.

- **Backtracking/tracing**: we find a point until the program is still OK, and try to isolate the location with a debugger or print statements.

- Testing and debugging go hand in hand. “test driven development”: don’t create more than one bug at a time.
The Object Oriented Paradigm

- Data oriented paradigm focuses on data and their representation in the computer.
- Process oriented approach focuses on the actions a software performs. Data are less important.
- In object oriented design data and functions acting on the data (methods) are equally important.
- Objects combine data and methods which operate on the data providing abstraction and encapsulation.
Abstraction

- The main idea is suppressing irrelevant details.
- Procedural abstraction: we focus on what we want to do without worrying about the details of how it is implemented.
- Data abstraction: we focus on objects and their interactions again without worrying about implementations.
- There are different levels of abstraction, as we move down in our top-down approach.
Encapsulation means hiding part of the information.

Conceptual independence: some details are hidden from the user, so he/she cannot modify those.

(As opposed to) Physical independence: an object really only depends on itself.

In Python, you can access most (even hidden) object data/variables, but it is not recommended.
Object oriented design deals with classification and relationship between objects.

- **Derivation**: new classes can be derived from existing classes
- **Inheritance**: new classes inherit properties of the parent classes.
- **Container**: an object that holds other objects. Searchable containers provide efficient searching facilities.
Object Hierarchies: Definitions Cont’d

- **Iterator**: provides a means by which objects within a container can be accessed one at a time
- **Visitor**: an operation to be performed on all objects in a container
- **Cursor**: represents the position of an object container
- **Adapter**: converts the interface of a class into an interface expected by the user.
- **Singleton**: an class which has only one instance.
There are many abstract data types (ADTs): e.g., stacks, queues, deques, ordered lists, sorted lists, hash and scatter tables, trees, priority queues, sets, and graphs. Most of them are represented as arrays and linked lists, which are called foundational data structures; both of them are collections.

- Array or list: an ordered collection of objects.
- Linked list: a collection of nodes, each of which has a container and a pointer possibly pointing to another node.
A Few Comments on **Python Lists**

- Python lists start at 0
- Slicing notation, e.g. `a[1:3]`
- Shallow copy–deep copy
- Tuple: immutable lists `(1, 2, 3)`
- Dictionaries: lists indexed by strings `{"a": 1, "b": 2}`
- Libraries also provide arrays
Example

```python
>>> a = [1, 2, 3]  # a python list
>>> b = a  # aliasing: shallow copy
>>> a[0]=0  # index start at 0
>>> b
[0, 2, 3]  # watch out!
>>> b = a[:]
# cloning: deep copy
>>> a[1]=1
>>> a
[0, 1, 3]
>>> b
[0, 2, 3]  # stays
```
Linked Lists are made of nodes linked to each other

- We can insert anywhere into a linked list without moving data.
- The end is usually marked with an empty link.
A python implementation of a linked list

class Node:
    def __init__(self, cargo=None, next=None):
        self.cargo = cargo
        self.next = next

    def __str__(self):
        return str(self.cargo)
Example: Creating a Linked List

```python
>>> node1 = Node(1)  # create
>>> node2 = Node(2)
>>> node3 = Node(3)
>>> node1.next = node2  # link
>>> node2.next = node3

>>> print node1      # print
1

Note that print changed in python 3
Stacks
They are the simplest of ADTs

Reminder: ADT is Abstract Data Type. When inventing algorithms it is often useful to think in terms of ADTs.

- LIFO: Last In First Out
- push: put something on the top of the stack
- pop: remove something from the top of the stack. The stack size is decreased, and always the last item is returned.
Python implementation

Python has direct support for stacks through lists

class Stack :
    def __init__(self) :
        self.items = []

    def push(self, item) :
        self.items.append(item)

    def pop(self) :
        return self.items.pop()

    def isEmpty(self) :
        return (self.items == [])
Example

Postfix Calculator

Postfix form is when the operator is after the operands. The other two possibilities are infix and prefix.

```python
>>> print evalPostfix("56 47 + 2 *")
206
```

For simplicity we allow integers and just two operations + and *.
Note: re is a built-in regular expression module in python, [^0-9] means “everything that is not a number”.

def evalPostfix(expr):
    import re
    tokenList = re.split("([^0-9])", expr)
    stack = Stack()
    for token in tokenList:
        ...

Example

Postfix Calculator: the split
Example

Postfix Calculator: using a stack

```python
if token == '' or token == ' ':
    continue
if token == '+':
    sum = stack.pop() + stack.pop()
    stack.push(sum)
elif token == '*':
    product = stack.pop() * stack.pop()
    stack.push(product)
else:
    stack.push(int(token))
return stack.pop()
```
Queues (and Dequeues)

- Queues embody the concept of “First come first served”, hence the name
- AKA First In First Out (FIFO’s): the first thing you put in comes out first
- The main difference from stack is that the removal happens at the opposite end as the insertion.
- Direct Python support with `pop(0)`
- Dequeues (double ended queues) are a generalization where you can insert and remove elements at both end
- Dequeues can be considered as a general ADT, of which stack and queue are special cases.
class Queue:
    def __init__(self):
        self.items = []

    def insert(self, item):
        self.items.append(item)

    def remove(self):
        return self.items.pop(0)

    def isEmpty(self):
        return (self.items == [])
Example

```python
>>> q=lecture2.Queue()
>>> q.insert("jill")
>>> q.insert("joe")
>>> q.insert("jane")
>>> print q.items
['jill', 'joe', 'jane']
>>> while len(q.items) > 0: print q.remove()
jill
joe
jane
```
Summary

- Testing and Debugging
- Elements of Object Oriented Design
- Basics of Data Structures
- A python list can serve as a stack, (de)queue
Homework # 2

- Reading: read chapters 5-9 of the python tutorial at http://http://docs.python.org/2/tutorial/.

- (E1) In a list of pairs of people (Joe, Jane) means that Joe is a relative of Jane. Write a program, which can calculate the number of groups, within which everybody is related (hint: note the similarity with the previous problem). Test, debug, and document your code according to the guidelines given in this chapter; this is understood for all programming assignments from now on.

- (E2) Write two python classes StackLL, QueueLL, in which the stack and queue data structures are realized with linked lists instead of arrays.
Homework # 2
continued

(E3) Write a Python program which realizes a dequeue data structure, where you can insert and remove elements at both ends of the queue (you are free to realize it any way you want to).

(E4) Write a program which takes an postfix expression, and rewrites it into prefix (hint: use a stack of strings).