

The *abalist* program: computing American Birding Association list totals

Zoological
Data Processing

John W. Shipman

2012-02-29 11:58

Abstract

Describes a system for counting birds according to the rules of the American Birding Association.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to john@nmt.edu.

Table of Contents

1. Introduction	2
1.1. How to get this publication	2
2. Operation of <i>abalist</i>	2
3. What is meant by ABA countable?	3
4. Internals of <i>abalist</i>	4
4.1. Prologue	5
4.2. class Args : Command line arguments	5
4.3. class FirstRecordSet	7
4.4. Class prologue	8
4.5. FirstRecordSet.addSight()	9
4.6. FirstRecordSet.purgeUncountables()	9
4.7. FirstRecordSet.__purgeCheck()	10
4.8. FirstRecordSet.__purgeAncestors()	11
4.9. FirstRecordSet.__removeSight()	11
4.10. FirstRecordSet.__promoteForm()	12
4.11. FirstRecordSet.genPhylo()	13
4.12. FirstRecordSet.genByDate()	13
4.13. FirstRecordSet.__init__()	13
4.14. class FirstSight : First-sighting object	14
4.15. addAllSightings() : Read the input files	15
4.16. addFile : Read one sightings file	15
4.17. addSighting() : Filter and add one sighting	16
4.18. filterOut() : Reject records by date and location	18
4.19. writeReports() : Generate final reports	18
4.20. Main program	19

¹ <http://www.nmt.edu/~shipman/aba/abalist/>

² <http://www.nmt.edu/~shipman/aba/abalist/abalist.pdf>

1. Introduction

The American Birding Association (ABA) sanctions competitive bird listing: who has seen the most bird species? There are numerous lists.

- The most inclusive is the world life list: who has seen the most bird species anyplace, ever?
- Some lists are constrained by location: who has seen the most species in North America, or one state?
- Some lists are constrained by time: who has seen the most birds in a calendar year?

There are combinations of time and space constraints as well: state year lists, day lists seen from a point location, and so on.

The purpose of the *abalist* program is to compute a fairly standard set of list totals: life lists, year lists, state lists, and state year lists.

This program depends on the author's sizeable infrastructure for managing bird records. Relevant links:

- *A system for representing bird taxonomy*³: Files describing bird classification and a system of short bird codes. We will refer to this as the *xnomo* system.
- *A system for encoding bird field notes*⁴: Covers the encoding of bird field observations for scientific purposes and a system for putting them up on the Web. In later sections we'll refer to this as the *birdnotes* system.

The *abalist* program reads the encoded field notes encoded for scientific use and interprets them according to the ABA's guidelines for countability.

1.1. How to get this publication

This publication is available in Web form⁵ and also as a PDF document⁶. Please forward any comments to **tcc-doc@nmt.edu**.

Files mentioned herein are available online:

- `abalist.py`⁷: The principle script.
- `sysargs.py`⁸: Author's command line argument digester module.
- For the `txny.py` module, see *A system for representing bird taxonomy*⁹.
- For the `birdnotes.py` module, see *A system for encoding bird field notes*¹⁰.

2. Operation of *abalist*

To compute list totals *abalist*, use this command:

```
abalist [-s stateCode] [-y YYYY] file ...
```

³ <http://www.nmt.edu/~shipman/xnomo/>

⁴ <http://www.nmt.edu/~shipman/aba/doc/>

⁵ <http://www.nmt.edu/~shipman/aba/abalist/>

⁶ <http://www.nmt.edu/~shipman/aba/abalist/abalist.pdf>

⁷ <http://www.nmt.edu/~shipman/aba/abalist/abalist.py>

⁸ <http://www.nmt.edu/~shipman/aba/abalist/sysargs.py>

⁹ <http://www.nmt.edu/~shipman/xnomo/>

¹⁰ <http://www.nmt.edu/~shipman/aba/doc/>

stateCode

To constrain the list to records from one state, supply a U.S. postal code argument. Example: “-s az” would count only Arizona records. If this option is omitted, all states are counted.

YYYY

To constrain the list to records from a calendar year, supply a four-digit year. Example: “-y 2004”
If this option is omitted, all years are counted.

file

List one or more XML files containing the field records to be scanned for countable ABA species. These files must conform to the **birdnotes.rnc** schema, as described in the *birdnotes* system.

The program writes two reports:

- The *chronological report* lists all countable records in ascending order by date. Multiple records for the same date are presented in phylogenetic order.
- The *phylogenetic report* lists all countable records in the phylogenetic order defined by the *xnomo* system.

3. What is meant by ABA countable?

The ABA has a large set of rules for which bird observations are countable. The observer is responsible for interpreting and applying these rules. The way records are coded in *birdnotes* files tells us these things about observations that are important to the *abalist* program:

- Their taxonomic identity, that is, what species, genus, or other taxon was observed. The *xnomo* system allows observations to be pegged to all taxonomic ranks, including a rank deeper than species that we call *form* rank. Forms include taxonomic subspecies, color morphs, and other fine distinctions within species.
- Whether the observer was certain of the identification. If the identification is in question, the observer codes the record with the **q='?'** attribute.
- Whether the observed individual is part of a countable population. For example, Whooping Cranes were regular in New Mexico for a time in the 1970s-1990s, but they were not an established breeding population, and hence not countable under ABA rules. The observer flags such records with the **q='-'** attribute in the *birdnotes* system.

Although the ABA totals are generally referred to as “species totals,” they may include observations pegged to other taxonomic ranks:

- Observations at the form level are counted as their containing species. For example, if either of both of Myrtle and Audubon's Warblers were observed, those count as their containing species, Yellow-rumped Warbler.
- Observations at higher ranks than species can be counted only if there are no observations pegged to lower ranks inside them. For example, if a list includes both “goose sp.” and Snow Goose, that counts as only one species. But if the observer reported “goose sp.” and saw no other kind of goose, that counts as one species.

It is also of interest to the author which observation was the first for a given species or other taxon. So, after assembling a list of observed taxa, here are the rules for counting the “species” in that list. We step through the list in phylogenetic order and apply these transformations to the tree of observed taxa:

1. If a form was observed but its parent species was not, promote the observation to the species level.

2. If a form was observed but its parent species was also observed, retain whichever observation was earlier, and peg it to the species level.
3. For any taxon at species level and higher, remove all ancestor taxa. For example, if there is an observation pegged to a given species, remove any observation from its containing genus, subfamily, family, etc.

After the tree is transformed by these rules, the number of nodes remaining is the species count for ABA purposes.

Here are some examples. In this table, the left-hand column shows the raw observations, and the right-hand column shows the transformed observation set.

Before	After
1999-12-27 American Green-winged Teal	1999-12-27 Green-winged Teal
2002-11-22 Snow Goose 2004-01-13 Blue Goose 2003-12-28 white Snow Goose	2002-11-22 Snow Goose
2004-01-13 Blue Goose 2003-12-28 white Snow Goose	2003-12-28 Snow Goose
1998-05-10 <i>Aechmophorus</i> sp. 2003-06-07 Western Grebe 2001-08-18 Clark's Grebe	2003-06-07 Western Grebe 2001-08-18 Clark's Grebe

In the first example, American Green-winged Teal is a form of the species Green-winged Teal. Since its parent species is missing, it is promoted to species.

The second example has Snow Goose and two of its forms. Because the species-level observation is first, it takes precedence over its forms.

In the third example, the white Snow Goose record is promoted to species level because it predates the other form.

The fourth example shows genus *Aechmophorus* and two of its species, Western and Clark's grebes. The genus is deleted because there are records below it.

4. Internals of *abalist*

The actual Python code follows, in “literate programming” style. For more on literate programming and the tool used in this application, see *A source extractor for lightweight literate programming*¹¹.

¹¹ <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/>

4.1. Prologue

We start the Python script with a line to make it self-executing, followed by a brief pro-forma comment.

```
#!/usr/bin/env python
#=====
# abalist: American Birding Association species totals report.
#
# For documentation, see:
#   http://www.nmt.edu/~shipman/aba/abalist/
#-----
```

abalist.py

Next comes a section of module imports.

```
#=====
# Imports
#-----

#--
# Python standard modules
#--

from __future__ import generators      # We use generators
import sys                             # Standard streams

#--
# From the author's Python library
#--

import sysargs                         # Command line argument processing

#--
# Application-specific modules
#--

from txny import *                    # Bird taxonomy & code system
from birdnotes import *              # For reading birdnotes files
```

abalist.py

4.2. class Args: Command line arguments

It is convenient to centralize processing of command line arguments in a class, and that is the purpose of class **Args**. It uses the author's **sysargs.py** module to combine processing of the arguments with automatic generation of a “usage message” showing the legal arguments. See the page on the author's Cleanroom library¹² for this module.

Before the **Args** class, we'll declare some manifest constants containing descriptions of *abalist's* command line arguments in the form required by the **sysargs.py** module.

```
#=====
# Manifest constants
#-----
```

abalist.py

¹² <http://infohost.nmt.edu/~shipman/soft/clean/lib.html>

```

#--
# Declarations for command line arguments
#--

STATE_OPTION = "s"          # Restrict by state
YEAR_OPTION  = "y"          # Restrict by year
FILES_ARG    = "files"      # List of input files

switchSpecs = [
    sysargs.SwitchArg ( STATE_OPTION,
        [ "Count only one state, e.g., '-s pa' " ],
        takesValue=1 ),
    sysargs.SwitchArg ( YEAR_OPTION,
        [ "Count only one year, e.g., '-y 1997' " ],
        takesValue=1 ) ]

posSpecs = [
    sysargs.PosArg ( FILES_ARG,
        [ "Names of input files in birdcodes format" ],
        repeated=1 ) ]

```

Now we're ready for the **Args** class. Here are the exported methods and attributes:

abalist.py

```

# - - - - - c l a s s   A r g s   - - - - -

class Args:
    """Represents all command line arguments.

    Exports:
    Args():
        [ if the command line arguments are valid ->
          return a new Args object representing those
          arguments
          else ->
            sys.stderr += (usage message) + (error message)
            stop execution ]
    .state:
        [ if no state option was selected -> None
          else -> the state code, upshifted ]
    .year:
        [ if no year option was selected -> None
          else -> the year as a string ]
    .fileNameList:
        [ list of file name arguments ]
    """

```

The constructor uses the **sysargs** module for overall argument checking, then stores the process arguments in its exported attributes.

abalist.py

```

# - - -   A r g s . _ _ i n i t _ _   - - -

```

```

def __init__ ( self ):
    """Constructor for the Args object."""

    #-- 1 --
    # [ if the command line arguments have options described
    #   by switchSpecs and positional arguments described by
    #   posSpecs ->
    #   sysArgs := a new SysArgs object representing them
    #   else ->
    #   sys.stderr += (usage message) + (error message)
    #   stop execution ]
    sysArgs = sysargs.SysArgs ( switchSpecs, posSpecs )

    #-- 2 --
    # [ self.state      := state code from sysArgs, upshifted,
    #   or None if not supplied
    #   self.year       := year string code from sysArgs, or
    #   None if not supplied
    #   self.fileNameList := positionals from sysArgs ]
    self.state      = sysArgs.switchMap [ STATE_OPTION ]
    if self.state is not None:
        self.state = self.state.upper()
    self.year       = sysArgs.switchMap [ YEAR_OPTION ]
    self.fileNameList = sysArgs.posMap [ FILES_ARG ]

```

4.3. class FirstRecordSet

The purpose of this object is to accumulate a set of first records of countable species (or higher taxa with no children, which count as species).

We use the **Sighting** object, from the `birdnotes.py` module, to represent each first sighting. This object has the four things we need to know: the kind of bird (as a **BirdId** object from the `abbr.py` module), the date, the state, and the name of the locality.

With the goal of being able to regurgitate the set of first records either in phylogenetic order or by date, the obvious choice for internal storage structures are two dictionaries:

- Private attribute `.__phyloMap` is a dictionary whose values are **Sighting** objects. The key for each value is the `.txKey` or taxonomic key string that we can use to sort the sightings into phylogenetic order.
- Private attribute `.__dateMap` also has **Sighting** objects as its values, but in this case the key is a 2-tuple consisting of the date string (e.g., "2005-01-30") and the taxonomic key. Sorting on this 2-tuple groups records with the same date, but within one date the records are in phylogenetic order.

So the **FirstRecordSet** object works like this:

1. We pour **Sighting** objects into it, and it keeps the first one for each taxon.
2. It has two methods for getting the sightings back out, one in phylogenetic order (`.genPhylo()`) and one in date order (`.genByDate()`).

4.3.1. Eliminating uncountable taxa

As described so far, the object will accumulate a lot of first sightings of taxa that turn out not to be countable. For example, if we register a sighting for “hawk sp.” and later register a sighting for Red-tailed Hawk, the first sighting isn't countable.

As described above, we can reduce the set of sightings to just the countable ones; see Section 3, “What is meant by ABA countable?” (p. 3).

When we add a new sighting to the set, we could immediately transform the tree in the ways discussed: promote subspecies to species, and purge higher taxa above sightings of lower taxa.

However, and forgive my possibly premature micro-optimization here, with large sets of sightings, this is likely to lead to a lot of processing every time a sighting is added. So, for performance reasons, we propose this **tradeoff**:

Instead of transforming the tree every time a new sighting is added, let's just leave them all in there, and require that the user call a “purge” method before generating the final reports. This method, called **.purgeUncountables()**, can take care of all the transformations in one pass through the set of taxa in the object.

If that method is not called before generating the final report, the report may include a number of uncountable taxa, as well as the first records for forms within a species.

4.4. Class prologue

Now to the actual code of the class.

abalist.py

```
# - - - - - c l a s s   F i r s t R e c o r d S e t   - - - - -
class FirstRecordSet:
    """Represents the set of first sightings for each taxon.

    Exports:
    FirstRecordSet ( txny ):
        [ txny is a Txny object containing the bird code system ->
          return a new, empty FirstRecordSet using that system ]
    .txny:          [ as passed to constructor, read-only ]
    .addSight ( sight ):
        [ sight is a first sighting as a FirstSight object
          if self contains no sighting for sight.txKey ->
            self := self + sighting
          else -> I ]
    .purgeUncountables():
        [ self := self with uncountable sightings removed ]
    .genPhylo():
        [ generate sightings in self in phylogenetic order,
          as FirstSight objects ]
    .genByDate():
        [ generate sightings in self by date,
          as FirstSight objects ]

    State/Invariants:
    .__phyloMap:
        [ dictionary whose values are FirstSight objects, and
```

```

        each key is the .txKey for that sighting ]
    ..__dateMap:
        [ dictionary whose values are FirstSight objects, and
          each key is (.date, .txKey) ]
    """

```

4.5. FirstRecordSet.addSight()

To add a sighting, we need to extract its date and taxonomic key for our internal dictionaries. Then, if this is the first sighting of this taxon, or the previously sighting isn't as early, we add the sighting to our dictionaries.

abalist.py

```

# - - - F i r s t R e c o r d S e t . a d d S i g h t - - -

def addSight ( self, sight ):
    """Add a new sighting to self, if it is a first."""

    #-- 1 --
    # [ if (self.__phyloMap has no key equal to sight.txKey) or
    #   (the sighting in self.__phyloMap[sight.txKey] is newer than
    #   sight) ->
    #   self.__phyloMap[txKey] = sight
    #   self.__dateMap[(date,txKey)] = sight ]
    try:
        otherSight = self.__phyloMap[sight.txKey]
        otherDate = otherSight.date
    except KeyError:
        otherDate = "9999-99-99"
    if sight.date < otherDate:
        self.__phyloMap[sight.txKey] = sight
        self.__dateMap[(sight.date, sight.txKey)] = sight

```

4.6. FirstRecordSet.purgeUncountables()

As describe in Section 3, “What is meant by ABA countable?” (p. 3), purging uncountable taxa from the tree uses these rules: forms are promoted to species unless there is an earlier record for the species, and higher taxa above are purged if there is any record below them.

Consequently, this method walks the tree by iterating over the keys of **self.__phyloMap**, applying this procedure for each sighting.

abalist.py

```

# - - - F i r s t R e c o r d S e t . p u r g e U n c o u n t a b l e s

def purgeUncountables ( self ):
    """Reduce the set of sightings to only the countable ones.

    NB: This might be modified to add a 'cutDepth=None'
    argument to allow counting of genera, forms, etc.
    """

```

```

#-- 1 --
# [ txKeyList := list of all keys in self.__phyloMap, sorted
#   cutDepth := rank depth of species rank
txKeyList = self.__phyloMap.keys()
txKeyList.sort()
cutDepth = self.txny.hier.speciesRank().depth

#-- 2 --
# [ self := self transformed to remove uncountable taxa ]
for txKey in txKeyList:
    #-- 2 body --
    # [ self := self with the sighting for txKey transformed
    #   as per the stated algorithm ]
    self.__purgeCheck ( txKey, cutDepth )

```

4.7. FirstRecordSet.__purgeCheck()

This method handles the purge transformation for one sighting.

abalist.py

```

# - - - F i r s t R e c o r d S e t . _ _ p u r g e C h e c k - - -

def __purgeCheck ( self, txKey, cutDepth ):
    """Purge or promote one sighting.

    [ (txKey is a key in self.__phyloMap) and
      (cutDepth is the depth of the countable rank) ->
        self := self with the sighting for txKey transformed
              as per the stated algorithm ]
    """

    #-- 1 --
    # [ taxon := the Taxon object for self.__phyloMap[txKey]
    #   sight := self.__phyloMap[txKey] ]
    sight = self.__phyloMap[txKey]
    taxon = self.txny.lookupTxKey ( txKey )

    #-- 2 --
    # [ if taxon.rank.depth <= cutDepth ->
    #   self := self with ancestors of sighting removed
    #   return
    # else if ((there is no sighting at depth=cutDepth that is
    #           an ancestor of sighting) or
    #           (sighting is earlier than that ancestor)) ->
    #   self := self with sighting promoted to cutDepth
    # else -> I ]
    if taxon.rank.depth <= cutDepth:
        self.__purgeAncestors ( txKey )
    else:
        self.__promoteForm ( txKey, cutDepth )

```

4.8. FirstRecordSet.__purgeAncestors()

The sighting whose taxonomic key is **txKey** takes precedence over any of its containing higher taxa, so purge all those higher taxa from self.

abalist.py

```
# - - -   F i r s t R e c o r d S e t . _ _ p u r g e A n c e s t o r s

def __purgeAncestors ( self, txKey ):
    """Remove all taxa containing txKey."""

    #-- 1 --
    # [ ancestorList := set of all taxonomic keys for
    #               ancestors of txKey ]
    taxon = self.txny.lookupTxKey ( txKey )
    ancestorList = []
    while taxon.parent is not None:
        ancestorList.append ( taxon.parent )
        taxon = taxon.parent

    #-- 2 --
    # [ self := self with any sightings matching
    #       ancestorList removed ]
    for ancestor in ancestorList:
        #-- 2 body --
        # [ ancestor is a Taxon in self.txny ->
        #   if self.__phyloMap has an entry whose txKey
        #   matches ancestor.txKey ->
        #     self.__phyloMap := self.__phyloMap with
        #                       that entry removed
        #     self.__dateMap := self.__dateMap with
        #                       that entry removed ]
        ancestorKey = ancestor.txKey
        if self.__phyloMap.has_key(ancestorKey):
            self.__removeSight ( ancestorKey )
```

4.9. FirstRecordSet.__removeSight()

This utility method removes a sighting from **self**, given its taxonomic key **txKey**. We have to pull the date out of the sighting before removing it, so we can remove it from the **.__dateMap** dictionary as well as **.__phyloMap**.

abalist.py

```
# - - -   F i r s t R e c o r d S e t . _ _ r e m o v e S i g h t   - - -

def __removeSight ( self, txKey ):
    """Remove one sighting from self.

    [ txKey is a key in self.__phyloMap ->
      self := self with that entry removed from
      self.__phyloMap and self.__dateMap ]
    """

    #-- 1 --
```

```

# [ sight := the sighting for txKey ]
sight = self.__phyloMap[txKey]

#-- 2 --
del self.__phyloMap[txKey]
del self.__dateMap[(sight.date, txKey)]

```

4.10. FirstRecordSet.__promoteForm()

This handles the form-promotion part of the purge algorithm.

abalist.py

```

# - - - F i r s t R e c o r d S e t . _ _ p r o m o t e F o r m - - -

def __promoteForm ( self, formKey, cutDepth ):
    """Check to see if a form sighting predates a species sighting.

    [ (formKey is the taxonomic key of a form in self) and
      (cutDepth is the depth of species rank) ->
        if ((there is no sighting at depth=cutDepth that is
            an ancestor of sighting) or
            (sighting is earlier than that ancestor)) ->
            self := self with sighting promoted to cutDepth
        else -> I ]
    """

    #-- 1 --
    # [ speciesKey := taxonomic key of the species that includes
    #               formKey
    #   formSight := self.__phyloMap[formKey] ]
    formSight = self.__phyloMap[formKey]
    ancestorTaxon = self.txny.lookupTxKey ( formKey )
    while ( ( ancestorTaxon.parent is not None ) and
            ( ancestorTaxon.rank.depth > cutDepth ) ):
        ancestorTaxon = ancestorTaxon.parent
    speciesKey = ancestorTaxon.txKey

    #-- 2 --
    # [ if ( ( self has an entry for speciesKey ) and
    #       ( that entry predates formSight ) ) ->
    #   delete formSight
    #   return
    # else if ( ( self has an entry for speciesKey ) and
    #          ( that entry doesn't predate formSight ) ) ->
    #   self := self with that entry deleted )
    # else -> I ]
    try:
        speciesSight = self.__phyloMap[speciesKey]
        if speciesSight.date <= formSight.date:
            self.__removeSight ( formKey )
            return
        else:
            self.__removeSight ( speciesKey )

```

```

except KeyError:
    pass

#-- 3 --
# [ newSight := a new FirstSight with txKey=speciesKey
#       and its other attributes taken from formSight
#   self := self with entry for formKey deleted ]
newSight = FirstSight ( speciesKey, formSight.date,
                       formSight.state, formSight.locName )
self.__removeSight ( formKey )

#-- 4 --
# [ self has an entry for speciesKey ->
#   self := self with entry for speciesKey deleted ]
self.__phyloMap[speciesKey] = newSight
self.__dateMap[(newSight.date, speciesKey)] = newSight

```

4.11. FirstRecordSet.genPhylo()

Generates the sightings in self using the keys of **self.__phyloMap**.

abalist.py

```

# - - -   F i r s t R e c o r d S e t . g e n P h y l o   - - -

def genPhylo ( self ):
    """Generate sightings in phylogenetic order."""
    txKeyList = self.__phyloMap.keys()
    txKeyList.sort()
    for txKey in txKeyList:
        yield self.__phyloMap[txKey]
    raise StopIteration

```

4.12. FirstRecordSet.genByDate()

Generates the sightings in self using the keys of **self.__dateMap**.

abalist.py

```

# - - -   F i r s t R e c o r d S e t . g e n B y D a t e   - - -

def genByDate ( self ):
    """Generate sightings in phylogenetic order."""
    keyList = self.__dateMap.keys()
    keyList.sort()
    for key in keyList:
        yield self.__dateMap[key]
    raise StopIteration

```

4.13. FirstRecordSet.__init__()

The constructor for **FirstRecordSet** stores its arguments and sets up its internal dictionaries.

```
# - - -   F i r s t R e c o r d S e t . _ _ i n i t _ _   - - -

def __init__ ( self, txnyn ):
    """Constructor for FirstRecordSet."""
    self.txnyn      = txnyn
    self.__phyloMap = {}
    self.__dateMap  = {}
```

4.14. class FirstSight: First-sighting object

When I first started to write this program, my thought was to use the **Sighting** objects to represent first sightings inside the **FirstSightingSet** object.

However, this ran into problems. The first problem was that the **Sighting** object contains a lot of stuff we don't care about in this application, so the **FirstSightingSet** class has to know a lot about the relationship of the **Sighting** object with the other objects in the *birdnotes* system.

One insidious little problem in particular led to the use of a more lightweight, single **FirstSight** object to represent sightings. The algorithm for finding out which sightings are countable (see Section 3, "What is meant by ABA countable?" (p. 3)) requires that in some cases forms (e.g., Myrtle Warbler) be promoted to species (e.g., Yellow-rumped Warbler). However, that would require the fabrication of a new **Sighting** object with a different taxonomic rank. In the **Sighting** object, bird identity is represented as a **BirdId** object, and that object requires a **fullAbbr** string in its constructor. But when we're promoting a form taxon to fill a species slot and there is no sighting already in the species slot, there's no obvious way to come up with the **fullAbbr** string for the species.

We could assume that all species have bird codes, but the author was hoping this program might someday be useful for counting ranks such as genera or families that might not have bird codes.

So, dealing with the transformation of **Sighting** objects just got too messy. All we really need to deal with in this application is four items: taxonomic key (as a stand-in for taxon), date, state, and locality name. All these are fairly easy to derive from a **Sighting** object, and transforming the **FirstSightingSet** object to remove uncountable forms becomes much easier. When we promote a form to species, we just change the **txKey** field to that of the species, plug it into the dictionaries, and we're done.

So, here's the small object **FirstSight** that carries these four attributes:

```
# - - - - -   c l a s s   F i r s t S i g h t   - - -

class FirstSight:
    """Represents the first sighting of some bird taxon.

    Exports:
    FirstSight ( txKey, date, state, locName ):
        [ (txKey is a taxonomic key string) and
          (date is a date as "YYYY-MM-DD") and
          (state is a state code, uppercased) and
          (locName is a locality name) ->
          return a new FirstSight object with those values ]
    .txKey:      [ as passed to constructor, read-only ]
    .date:       [ as passed to constructor, read-only ]
    .state:      [ as passed to constructor, upshifted, read-only ]
```

```

        .locName:      [ as passed to constructor, read-only ]
        """

# - - -   F i r s t S i g h t .   _ _   i n i t   _ _   - - -

def __init__ ( self, txKey, date, state, locName ):
    """Constructor for FirstSight"""
    self.txKey    = txKey
    self.date     = date
    self.state    = state.upper()
    self.locName  = locName

```

4.15. addAllSightings (): Read the input files

This function iterates over all the file names to be read, adding the contents of each file to the **FirstRecordSet** object we were passed as an argument.

abalist.py

```

# - - -   a d d A l l S i g h t i n g s   - - -

def addAllSightings ( args, txny, firstSet ):
    """Read all input files.

    [ (args is an Args object) and
      (txny is a Txny object) and
      (firstSet is a FirstSightingSet) ->
      firstSet := firstSet with sightings added from files named
                  in args.fileList, filtered by the selection criteria in
                  args, using txny as the name authority
      sys.stderr += messages about undefined location codes in
                  those files, if any ]
    """
    #-- 1 --
    for fileName in args.fileNameList:
        #-- 1 loop --
        # [ fileName is a string ->
        #   if fileName names a readable, valid birdnotes file ->
        #     firstSet := firstSet with countable sightings
        #               added from that file, filtered according to args
        #   sys.stderr += messages about undefined location codes
        #               in that file, if any ]
        addFile ( args, txny, firstSet, fileName )

```

4.16. addFile: Read one sightings file

We use the *birdnotes* system to read the sightings file and convert it into a **BirdNoteSet** object. Then we ravel out all the individual sightings from that object, and filter them according to the options in **args**.

abalist.py

```

# - - -   a d d F i l e   - - -

def addFile ( args, txny, firstSet, fileName ):

```

```

""""Read one input file.

[ (args is an Args object) and
  (txny is a Txny object) and
  (fileName is a string) and
  (firstSet is a FirstRecordSet) ->
  if fileName names a readable, valid birdnotes file ->
    firstSet := firstSet with countable sightings
              added from that file, filtered according to args
    sys.stderr += messages about undefined location codes
              in that file, if any ]
""""

#-- 1 --
# [ birdNoteSet := a new, empty BirdNoteSet using txny ]
birdNoteSet = BirdNoteSet ( txny )

#-- 2 --
# [ if fileName names a readable XML file conforming to
#   birdnotes.rnc ->
#   birdNoteSet := birdNoteSet with all data added from
#                 that file ]
birdNoteSet.readFile ( fileName )

#-- 3 --
# [ firstSet := firstSet with all sightings added from
#   birdNoteSet, filtered according to args ]
for dayNotes in birdNoteSet.genDays():
  for form in dayNotes.genForms():
    for sighting in form.genSightings():
      #-- 3 body --
      # [ if sighting is countable and passes the
      #   filtering according to args ->
      #   firstSet := firstSet with sighting added
      #   sys.stderr += messages about undefined
      #   location codes in sighting, if any
      #   else -> I ]
      try:
        addSighting ( args, firstSet, sighting )
      except KeyError, detail:
        print >>sys.stderr, (
          "*** Undefined location, %s: %s" %
            (dayNotes.date, detail) )

```

4.17. addSighting(): Filter and add one sighting

The **sighting** argument is a bird sighting as a **Sighting** from the *birdnotes* module. The purpose of this function is to implement the time and location filtering specified in the **args** object, and for those sightings that pass, convert them to a **FirstSight** object and add them to the **firstSet** object in which we're accumulating the filtered first sightings.

```

# - - -   a d d S i g h t i n g   - - -

def addSighting ( args, firstSet, sighting ) :
    """Filter and add sightings.

        [ (args is an Args object) and
          (firstSet is a FirstRecordSet object) and
          (sighting is a birdnotes.Sighting object) ->
            if sighting is countable and passes the filtering
            according to args ->
                firstSet := firstSet with a new sighting added made
                           from sighting
            else -> I ]
    """

    #-- 1 --
    # [ if sighting is uncountable because of status (q='?' or
    #   q='-') or because it is second hand (fide='...') ->
    #   return
    #   else -> I ]
    ageSexGroup = sighting.ageSexGroup
    if ageSexGroup:
        if ageSexGroup.fide:
            return
        if ageSexGroup.q:
            if ageSexGroup.q != ' ':
                return

    #-- 2 --
    # [ newSight := a FirstSight object made from sighting ]
    birdForm = sighting.birdForm
    txKey     = birdForm.birdId.taxon.txKey
    dayNotes = sighting.dayNotes
    date     = dayNotes.date
    state    = dayNotes.regionCode
    locName  = sighting.getLocGroup().loc.name
    newSight = FirstSight ( txKey, date, state, locName )

    #-- 3 --
    # [ if sighting passes the filtering according to args ->
    #   I
    #   else -> return ]
    if filterOut ( args, newSight ) :
        return

    #-- 4 --
    # [ firstSet := firstSet with newSight added ]
    firstSet.addSight ( newSight )

```

4.18. filterOut(): Reject records by date and location

This function applies the filtering criteria specified in the **Args** class. Right now we only filter by state and year. Fancier stuff could be implemented later, and the only two places that would have to change are right here and in the **Args** class.

abalist.py

```
# - - -   f i l t e r O u t   - - -

def filterOut ( args, sight ):
    """Predicate: does this record get filtered out?

    [ (args is an Args object) and (sight is a FirstSight) ->
      if sight doesn't pass the filtering criteria of args ->
        return 1
      else -> return 0 ]
    """

    #-- 1 --
    # [ if ( ( args specifies year filtering ) and
    #       ( sight.date isn't in args.year ) ) ->
    #   return 1
    #   else -> 0 ]
    if args.year:
        year = sight.date[:4]
        if year != args.year:
            return 1

    #-- 2 --
    # [ if ( ( args specifies state filtering ) and
    #       ( sight.state doesn't match args.state ) ) ->
    #   return 1
    #   else -> return 0 ]
    if ( ( args.state ) and ( args.state != sight.state ) ):
        return 1
    else:
        return 0
```

4.19. writeReports(): Generate final reports

There are two reports: phylogenetic and chronological. Each is driven by a method on the **firstSet** object that generates all the records in a given order.

abalist.py

```
# - - -   w r i t e R e p o r t s   - - -

def writeReports ( txn, firstSet ):
    """Generate all output reports.

    [ firstSet is a FirstRecordSet ->
      sys.stdout += (phylogenetic report from firstSet) +
                   (chronological report from firstSet) ]
    """
```

```

#-- 1 --
# [ firstSet := firstSet with all uncountable forms purged ]
firstSet.purgeUncountables()

#-- 2 --
# [ sys.stdout += phylogenetic report from firstSet ]
print "=== Phylogenetic listing ==="
spCount = 0
for first in firstSet.genPhylo():
    spCount += 1
    taxon = txnny.lookupTxKey ( first.txKey )
    print("{count:4d}. {first.date} {eng} "
          "[{first.state}: {first.locName}]"
          .format(
            count=spCount, first=first, eng=taxon.eng))
####      print ( "%4d. %-32s %s\n                %s: %s" %
####          (spCount, taxon.eng, first.date, first.state,
####            first.locName ) )

#-- 3 --
# [ sys.stdout += chronological report from firstSet ]
print "=== Chronological listing ==="
spCount = 0
for first in firstSet.genByDate():
    spCount += 1
    taxon = txnny.lookupTxKey ( first.txKey )
    print("{count:4d}. {first.date} {eng} "
          "[{first.state}: {first.locName}]"
          .format(
            count=spCount, first=first, eng=taxon.eng))
####      print ( "%4d. %-32s %s\n                %s: %s" %
####          (spCount, taxon.eng, first.date, first.state,
####            first.locName ) )

```

4.20. Main program

In the main, first we check command line arguments by instantiating an **Args** object.

abalist.py

```

# - - - - -   m a i n   - - - - -

#-- 1 --
# [ if the command line arguments are valid ->
#   return a new Args object representing those arguments
#   else ->
#   sys.stderr += (usage message) + (error message)
#   stop execution ]
args = Args ()

```

Next, we read the taxonomic authority file so we know what all the bird codes mean and how they're classified.

abalist.py

```

#-- 2 --
# [ there is a readable, valid "aou.xml" file in this directory ->

```

```
#   txnny := a Txny object representing that file ]
txnny = Txny()
```

Next, we instantiate a **FirstRecordSet** object which will hold the first sightings for every taxon.

abalist.py

```
## 3 --
# [ firstSet := a new, empty FirstRecordSet object using txnny ]
firstSet = FirstRecordSet ( txnny )
```

Now we read all the input files, adding qualifying countable sightings to the **firstSet** object.

abalist.py

```
## 4 --
# [ firstSet := firstSet with countable sightings added from
#   files named in args.fileList, filtered by the selection
#   criteria in args
#   sys.stderr += messages about undefined location codes in
#   those files, if any ]
addAllSightings ( args, txnny, firstSet )
```

Finally, we dump out the accumulated first sightings into the two final reports.

abalist.py

```
## 5 --
# [ sys.stdout += (phylogenetic report from firstSet) +
#   (chronological report from firstSet) ]
writeReports ( txnny, firstSet )
```