

A system for encoding bird field notes

Zoological
Data Processing

John W. Shipman

2011-04-10 12:22

Abstract

This publication describes an XML schema for representing a birdwatcher's field notes, along with instructions for using an accompanying module written in the Python programming language.

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. Requirements	3
2. Why XML?	3
3. Design considerations	4
3.1. The time dimension	4
3.2. The spatial dimension	4
3.3. The taxonomic dimension: what kind of bird?	6
3.4. Other dimensions of the data	6
4. Entity-relationship diagram	7
5. The schema	7
5.1. The preamble	8
5.2. <code>note-set</code> : the root element	8
5.3. The <code>day-notes</code> element	8
5.4. The <code>day-summary</code> element: information about the field day	9
5.5. The <code>loc</code> element: Defining location codes	10
5.6. The <code>gps</code> element: defining a waypoint	11
5.7. The <code>day-annotation</code> elements	12
5.8. The <code>form</code> element: records for one kind of bird	12
5.9. The <code>taxon-group</code> pattern: biological classification of the birds seen	14
5.10. The <code>age-sex-group</code> attributes	15
5.11. The <code>loc-group</code> pattern: locality attributes and content	16
5.12. The <code>sighting-notes</code> elements	16
5.13. The <code>photo</code> element	17
5.14. The <code>floc</code> element: Multiple sightings of a given form	17
5.15. The narrative elements	18
6. Directory structure for data files	19
7. <code>birdnotes.py</code> : A Python interface	19
8. <code>class BirdNoteSet</code> : Container for notes	20
8.1. <code>BirdNoteSet()</code> : Constructor	20

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

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

8.2.	<code>BirdNoteSet.genDays()</code> : Generate contained daily sets	21
8.3.	<code>BirdNoteSet.addDay()</code> : Add a daily note set	21
8.4.	<code>BirdNoteSet.readFile()</code> : Read an XML file	21
8.5.	<code>BirdNoteSet.writeFile()</code> : Save as XML	21
9.	<code>class DayNotes</code> : Notes for one day	21
9.1.	<code>DayNotes.title()</code> : General title string	22
9.2.	<code>DayNotes.defaultLoc()</code> : What is the default location?	22
9.3.	<code>DayNotes.lookupLoc()</code> : Look up a location code	22
9.4.	<code>DayNotes.addForm()</code> : Add a new form record	22
9.5.	<code>DayNotes.genForms()</code> : Retrieve stored sightings	22
9.6.	<code>DayNotes.genFormsSeq()</code> : Retrieve sightings in addition order	23
10.	<code>class DaySummary</code> : Daily context	23
10.1.	<code>DaySummary.defaultLoc()</code> : What is the default location?	23
10.2.	<code>DaySummary.addLoc()</code> : Add a new locality code definition	23
10.3.	<code>DaySummary.lookupLoc()</code> : Look up a location code	23
10.4.	<code>DaySummary.genLocs()</code> : Retrieve all locality definitions	24
11.	<code>class Loc</code> : Locality code definition	24
11.1.	<code>Loc.addGps()</code> : Add a waypoint	24
11.2.	<code>Loc.genGps()</code> : Generate waypoints	24
12.	<code>class Gps</code> : GPS waypoint	24
13.	<code>class BirdForm</code> : Records for one kind of bird	25
13.1.	<code>BirdForm.addSighting()</code>	25
13.2.	<code>BirdForm.__len__()</code> : Return the number of sightings	25
13.3.	<code>BirdForm.__getitem__()</code> : Retrieve a child sighting	26
13.4.	<code>BirdForm.nSightings()</code> : Number of sightings	26
13.5.	<code>BirdForm.getLocGroup()</code> : Get the effective locality	26
14.	<code>class LocGroup</code> : Inheritable locality data	26
15.	<code>class SightNotes</code> : Sighting notes	26
15.1.	<code>SightNotes.addPara()</code> : Add general notes	27
15.2.	<code>SightNotes.addPhoto()</code> : Add a photo link	27
15.3.	<code>SightNotes.genPhotos()</code> : Retrieve photo links	27
16.	<code>class Sighting</code> : Single sighting	27
16.1.	<code>Sighting.getLocGroup()</code>	28
17.	<code>class AgeSexGroup</code> : Age and sex details	28
18.	<code>class Photo</code> : Photograph link	28
19.	<code>class Narrative</code> : General container for narrative	28
19.1.	<code>Narrative.addPara()</code> : Add a paragraph	29
19.2.	<code>Narrative.__len__()</code> : How many paragraphs?	29
19.3.	<code>Narrative.__getitem__()</code> : Get one paragraph	29
19.4.	<code>Narrative.genParas()</code> : Generate the contained paragraphs	29
20.	<code>class Paragraph</code> : General container for a paragraph of narrative	29
20.1.	<code>Paragraph.addContent()</code> : Add content	29
20.2.	<code>Paragraph.genContent()</code> : Generate the phrase list	30
21.	<code>class BirdNoteTree</code> : A complete set of notes	30
21.1.	<code>BirdNoteTree()</code> : Constructor	30
21.2.	<code>BirdNoteTree.genMonths()</code> : Generate monthly record sets	31
22.	<code>class FlatSighting</code> : Complete sighting record	32

1. Requirements

Formal field notes taken while birdwatching have some value to the scientific community. The system described here is primarily intended to archive such notes and make them accessible for study.

A secondary purpose is to serve the needs of recreational birdwatchers who want to know what others have seen so they can plan their own field trips.

This system does not address the needs of a system for storing the notes from multiple observers. The identity of the primary observer is not represented internally, although that observer can mention other observers who were present for some or all of the sightings.

At this writing, the primary thrust of the design effort is to produce an internal representation of bird notes that is accurate yet flexible and reasonably easy to use. The external rendering of the encoded notes is an open-ended challenge. At a minimum, the notes ought to be available on the Web in chronological order.

Future renderings will include an archival print form, and assorted facilities for querying records through the Web. Such queries will allow users to limit records by time period, locality, type of bird, and type of data. For example, a user might ask, what are all this observer's records for Vermilion Flycatcher? What records are there of the vocalizations of Verdin? What are the observer's American Birding Association totals for countable species for the state of New Mexico?

This document includes:

- A discussion of why XML was the chosen representation, and an XML schema that defines the document type used to represent the bird records.
- A programmatic interface in the Python language that can be used to access records in that XML document type.

The actual implementation of the Python interface is discussed in a companion publication, *bird-notes.py: Objects to represent bird note files*³.

2. Why XML?

Due to the structural complexity of bird notes, in the author's opinion XML is the best form in which to represent the notes internally.

The author also prefers the Relax NG Compact Format for representing the schema, or structure, of this XML document type. The reader is also referred to the XSLT programming language, which is used for the Web rendering.

References:

- XML and SGML: Using tagged text ⁴: General online help for XML.
- Relax NG Compact Syntax (RNC) ⁵: The author's preferred schema language.
- XSLT Reference ⁶: A language for simple, quick-and-dirty rendering of XML documents.
- XML document authoring with *emacs nxml -mode* ⁷: The author's preferred tool for creating valid XML documents.

³ <http://www.nmt.edu/~shipman/aba/doc/pyims>

⁴ <http://www.nmt.edu/tcc/help/xml/>

⁵ <http://www.nmt.edu/tcc/help/pubs/rnc/>

⁶ <http://www.nmt.edu/tcc/help/pubs/xslt/>

⁷ <http://www.nmt.edu/tcc/help/pubs/nxml/>

The author wishes to thank Dr. David Mundie for getting him started in the SGML family of markup languages, many years ago, while discussing techniques for the encoding of bird notes. The author has built numerous working XML applications since then and is glad to return to this problem with a much better appreciation of tools and techniques gained in the interim.

3. Design considerations

In any representation of the real world, there is always a tradeoff: what qualities are formalized, and which ones informal? When designing XML document types, we don't want every possible thing to result in a new kind of element. Some things are important enough to rate their own tags or attributes, but some things are best left as good old English text.

Similarly, there is a tradeoff in validation. The `nxml-emacs` package for the `emacs` text editor ensures that the XML document you create is *valid*, that is, that it conforms to the schema. But the schema can go only so far: a valid document may still have problems that prevent it from being useful. Ultimately a human should go over the rendered form of the information with a critical eye, and fix the problems manually. However, it may be possible to write additional software tools to check the content, and effort invested in such tools may pay off in the long run.

To be more specific, let's turn to the problem at hand. To represent sightings of birds, there are three fundamental dimensions.

3.1. The time dimension

Time: When was the bird seen? For our purposes, a granularity of one day is a good first approximation. Birders generally record the day of a sighting. However, there are a few cases where the time of day is important. For example, certain hummingbirds may come to feeders only at certain times or only at certain intervals. We don't need to formalize that information if bird records are allowed to contain generic, unstructured text: "The Carolina Wren generally stops singing here before 10am."

3.2. The spatial dimension

Space: Where was the bird seen? There is a lot of room for tradeoffs here. Ultimately it would be a fine thing to have a GIS (Geographic Information System), with its highly flexible mechanism for representing points and larger areas in space.

Setting up a GIS takes a lot of time and effort. What we need for a first approximation is a less formal system that will not create too many problems if the geography is formalized more in the future.

3.2.1. The locality hierarchy

For this system, we'll represent locality as three levels:

1. The *region code* is the code for the state or province. This code will be the U.S. postal code for U.S. states (e.g., NM for New Mexico). We won't worry about countries yet (since the author has done no birding abroad), but it should be straightforward to add region codes for foreign countries, and infer the nationality from the region code.
2. The *location name* is a single string describing the location within the region. It may be very broad (e.g., "Roosevelt Co." for an entire county) or more specific (the name of a town).

A good way to add more flexibility to location names is to use an informal but hierarchical structure, enumerating a series of areas from larger to smaller, separated by colons. Example: "Bosque del Apache: Headquarters". Three or more levels might be used.

- Sometimes you need to describe the locality very precisely. For this situation, we'll allow an optional *location detail* level, consisting of unstructured English text as needed. For example, "In the northeast corner there are some rusted oil drums that have standing water on them from the recent rains. The birds were coming in here to drink."

The region code is required, because both scientific and recreational consumers of the data need to know in what state the bird occurred. The location name will do most of the rest of the work for most of the cases, with location detail being added where necessary.

Because location names can be rather lengthy, we resort to a formalized shorthand code for the location names used within a single day's records. This *location code* is used to save typing in the bulk of records. For example, for the location name "Bosque del Apache NWR" we could use the code **BdA**. Such codes must start with a letter and may contain only letters, digits, hyphen ("-"), and underbar (_).

3.2.2. GPS coordinates

Coordinates obtained from a Geographical Positioning System (GPS) are extremely useful in nailing down the exact locality of bird sightings, road junctions, and such.

In order to represent GPS coordinates precisely with a minimum of fuss, we use an encoding system for waypoints discussed fully in *A Python mapping package*⁸.

Here's an excerpt from that document that describes what we call *flexible angle format*:

- Numbers with 1, 2 or 3 digits are assumed to be in degrees. For example, "34" is assumed to be 34°, and "107" means 107°.
- Numbers with 4 or 5 digits are assumed to be in the format *DDMM* or *DDDMM*, where *DD* or *DDD* is degrees and *MM* is minutes. So, for example, 3456 means 34° 56', and 10703 means 107° 3'.
- Numbers with 6 or 7 digits are assumed to have be in format *DDMMSS* or *DDDMMSS*, where *SS* is seconds. Examples: 341638 means 34° 16' 38", and 1075301 means 107° 53' 1".
- A trailing decimal point and fraction are always allowed. The fraction is assumed to be in the same units as the smallest unit to the left of the decimal point. For example, "3401.57" is interpreted as 34° 01.57'.

The syntax of a waypoint is:

```
lat{n|s} lon{e|w}
```

That is, start with the latitude using the flexible angle format described above, followed by *n* or *s* for north or south latitude, followed (optionally) by whitespace, followed by the longitude in flexible angle format, then *e* or *w* for east or west longitude.

Here are some examples:

5130s0007e	51°30' S. Lat., 0°7' E. Long.
34n 107w	34° N. Lat., 107° W. Long.
34.0242n 107.1811w	34.0242° N. Lat., 107.1811° W. Lat.

⁸ <http://www.nmt.edu/tcc/help/lang/python/mapping/doc/>

3.3. The taxonomic dimension: what kind of bird?

To represent what kind of birds we see, we're fortunate to have the American Ornithological Union's *Check-list of North American Birds* as a standard for the classification of birds.

However, the *AOU Check-List* is not a static entity. Every two years a new revision appears, and every few revisions a new major edition appears. At this writing, the Seventh Edition is the base checklist, with supplements through the Forty-Seventh Supplement adding later corrections. The author has built a complete infrastructure for tracking all these versions; it is described in *A system for representing bird taxonomy*⁹. The fundamental item used from this system is the six-letter bird code, such as VERFLY for Vermilion Flycatcher or BRWHAW for Broad-winged Hawk.

Fundamentally, we prefer to assign each sighting to a biological taxon in the AOU's classification. Most sightings are identified to species, but we may use larger or smaller taxa. For example, what in English we call "falcon sp." (falcon species) can be assigned to family Falconidae. In another case we might see a well-marked race such as Audubon's Warbler, which is a form of the species Yellow-rumped Warbler. The six-letter codes in the nomenclatural base system allow for assignment to any level: these examples would be encoded as FALCON and AUDWAR respectively.

Two additional wrinkles complicate the process of assigning sightings to taxa:

- Some observations can be narrowed down to two forms but no further. For example, it is often quite difficult to separate Hammond's Flycatcher and Dusky Flycatcher in the field. We refer to this situation as a *species pair*. Literature references might show this form as "Hammond's/Dusky Flycatcher."
- Some of the individuals out there are hybrids. In this case we can generally guess at the two parent forms, e.g., "Mallard x Gadwall".

3.4. Other dimensions of the data

Several other aspects of bird observations are worth recording. Here are some:

- Age and sex. For most purposes two age classes (adult and immature) suffice.
- Description. The rarer the record, the more skepticism will greet it. It's important in these cases for the observer to report on what basis the bird was identified. What did the plumage look like? What was its overall shape? The color of unfeathered parts such as bills and feet?
- Vocalization. The calls and songs birds make can be important in identification and are of interest in their own right.
- Behavior. The actions of birds are another area worth studying.
- Solidity of identification. Is the observer convinced that the identification of the kind of bird is correct?
- Whether the form is considered countable by the American Birding Association. For example, exotics are not countable until they establish a stable breeding population over ten years, so all the author's sightings of Whooping Crane in New Mexico are not countable.

The list may grow with time as this system of representation matures. One of the virtues of XML is that you can add new tag and attribute types to a document type as the needs change.

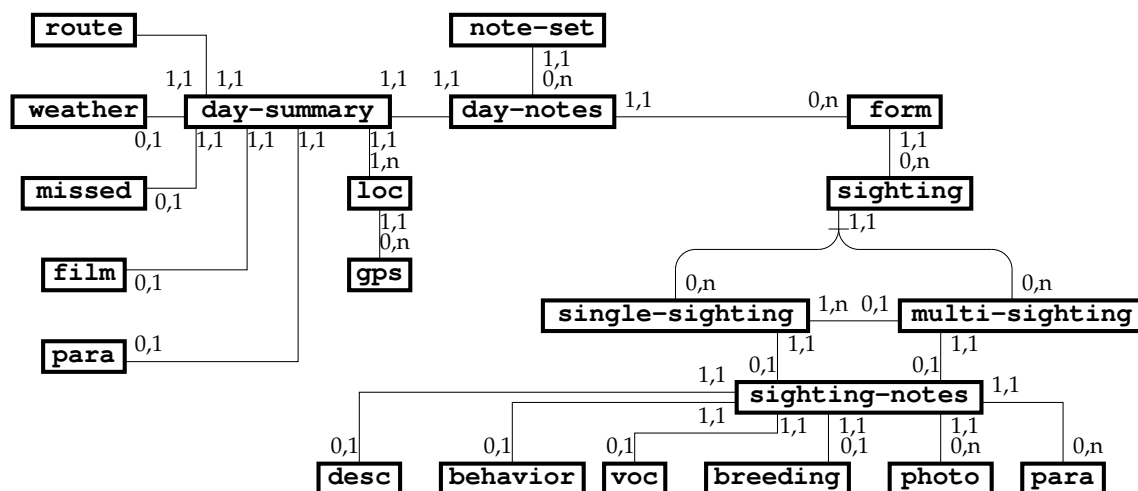
With these considerations in mind, let us turn to the actual XML schema.

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

4. Entity-relationship diagram

This diagram shows the relationship between the elements of the schema. The boxes represent entities. The lines between the boxes represent relationships. The numbers at each end of the connecting lines show the minimum and maximum number of entities at each end of the relationship.

For example, each document has a root element `note-set`. It contains zero or more `day-notes` elements, each of which represents a set of notes in one day (and in only one state). Each `day-notes` element pertains to one and only one `note-set`.



- Section 5.2, “`note-set`: the root element” (p. 8).
- Section 5.3, “The `day-notes` element” (p. 8).
- For `day-summary`, see Section 5.4, “The `day-summary` element: information about the field day” (p. 9). For the elements related to `day-summary`, see Section 5.7, “The `day-annotation` elements” (p. 12). For the `para` element, see Section 5.15, “The narrative elements” (p. 18).
- Section 5.5, “The `loc` element: Defining location codes” (p. 10).
- Section 5.6, “The `gps` element: defining a waypoint” (p. 11).
- Section 5.8, “The `form` element: records for one kind of bird” (p. 12) describes the `form` element and its related elements.

5. The schema

How should a large collection of bird notes be organized into files? The author feels that a month's worth of notes is a reasonable size for one file. Files are named as `yyyy-mm.xml`, where `yyyy` is the year and `mm` is the month number. The overall organization places a year's worth of notes in one directory whose name is the year number. So, from the root directory, file `2004/2004-09.xml` contains the notes for September 2004.

So to begin our discussion of the schema, we assume that each document represents all the notes for one calendar month.

5.1. The preamble

Here, in literate programming form, is the Relax NG Compact Syntax (RNC) schema for the document type that encodes the field notes. For more on the author's approach to literate programming, see *Lightweight literate programming*¹⁰.

First, a little preamble:

```
# birdnotes.rnc: Relax NG schema for bird field notes
# $Revision: 1.55 $ $Date: 2011/04/10 18:22:30 $
#-----
# For documentation, see:
# www.nmt.edu/~shipman/aba/doc/
#-----
```

birdnotes.rnc

5.2. note-set: the root element

As the root element of our bird notes XML document type, we choose `note-set`.

```
start = note-set

note-set = element note-set
{ attribute period { text }, 1
  day-notes* 2
}
```

birdnotes.rnc

- 1** This attribute should be the full month name, one space, and the four-digit year. This form of the date will appear in Web and print renderings of the notes. Example:

```
period=' June 2004'
```

- 2** For each day spent in the field, create a `day-notes` element. See Section 5.3, “The `day-notes` element” (p. 8).

If multiple states were worked in a single day, use a separate `day-notes` element for each state.

5.3. The day-notes element

Each `day-notes` element represents one day's effort within a single state. If multiple states are covered in a single day, each day's notes must be segregated into a separate `day-notes` element.

```
day-notes = element day-notes
{ attribute state {state-pattern}, 1
  attribute date {date-pattern}, 2
  attribute day-loc { xsd:Name }?, 3
  day-summary, 4
  form* 5
}
state-pattern = xsd:string{pattern="[a-z]{2,3}"}
date-pattern = xsd:string{pattern="[0-9]{4}-[0-9]{2}-[0-9]{2}"}
```

birdnotes.rnc

¹⁰ <http://www.nmt.edu/~shipman/soft/litprog/>

- 1 Defines the state or other region where its sightings occurred. The `state` and `date` attributes are required. The state code is case-insensitive, so it isn't necessary to capitalize it. Example:

```
state='nm'
```

- 2 This attribute specifies the date of the fieldwork. It must have the format `yyyy-mm-dd`, although unlike the `xsd:date` datatype, we allow the month or day to be `'00'`. This necessary because of vague dates like `'1974-11-00'` (sometime in November 1974) and `'1989-00-00'` (sometime in 1989). Example:

```
date='2004-09-05'
```

- 3 The optional `day-loc` attribute specifies a location code that describes a geographic area containing the entire day's efforts. The name related to this location code will be used as the brief title associated with that day's notes. If given, this attribute's location code must be defined by a `loc` element (described below).

If no `day-loc` attribute is supplied, the default location code will be the one given in the `default-loc` attribute of the `day-summary` element. See the discussion of the `default-loc` attribute under `day-summary`.

- 4 The content of a `day-notes` element must start with exactly one `day-summary` element; see Section 5.4, "The `day-summary` element: information about the field day" (p. 9).
- 5 The rest of the content consists of zero or more `form` children, one for each kind of bird seen. See Section 5.8, "The `form` element: records for one kind of bird" (p. 12).

5.4. The `day-summary` element: information about the field day

The first child of a `day-notes` element must be a `day-summary` element. This element describes all the general information about one day in the field (other than records of birds seen):

birdnotes.rnc

```
day-summary = element day-summary
{
  attribute default-loc { xsd:Name }, 1
  loc+, 2
  day-annotation 3
}
```

- 1 One `default-loc` attribute is required. This attribute specifies the default location code that should be used for bird records that don't specify their own location code. The code value must be defined in one of this element's `loc` children.

The reason `day-loc` is not always the same as the `default-loc` is that they often apply to different areas. The `day-loc` must include all areas birded that day, so that the indices and tables of contents enumerating daily notes can accurately describe the total area worked. The purpose of the `default-loc`, on the other hand, is to supply a default locality for the majority of records.

For example, suppose the observer saw 80 species of birds at Bosque del Apache Refuge (code "BdA"), but there were a few records from the town of Socorro (code "SOC") included. In that case, we would define a third location code "day" as "Socorro-Bosque del Apache", and encode the `day-notes` element with an attribute `day-loc='day'`, and the `day-summary` element would have an attribute `default-loc='BdA'`, which make it unnecessary to supply an explicit location code for all 80 of the Bosque records.

Note

This is an example of a validity condition that can't easily be specified in the schema: all references to location codes must be defined in a `loc` element for that day, but an RNC schema can't really stipulate that. Writing a separate software tool to validate this might be a worthwhile effort. See Section 3, "Design considerations" (p. 4) for a discussion of the validation tradeoffs.

- 2 The content starts with one or more `loc` elements that define all the location codes used on that day, followed by an optional assortment of other elements that are defined by the `day-annotation` pattern. See Section 5.5, "The `loc` element: Defining location codes" (p. 10).
- 3 Following the block of `loc` elements, you can include an assortment of other child elements in any order that further describe the day in the field. See Section 5.7, "The `day-annotation` elements" (p. 12).

Here's an example of a day-summary element that illustrates most of the features:

```
<day-summary default-loc='MT'>
  <loc code='day' name='Portales-Melrose Trap-Tatum' />
  <loc code='MT' name='Melrose Trap' />
  <loc code='ph' name='Clovis-Tatum route: nameless pothole'>
    <gps waypoint='335344n 1032010w' />
  </loc>
  <route>
    Stayed in Portales. Birded Melrose Trap 1110-1150, left
    due to windy conditions. Returned 1515-1630, incipient
    rain. Tried to find the Tatum sewage ponds to chase the
    rumored Yellow-crowned Night Heron, couldn't find them by
    the time it got too dark.
  </route>
  <film>
    Started a new roll of VC 400, not finished: 2004-09-03
    1900 (no shots) : 2004-09-05 1030.
  </film>
  <para>
    Drove into Melrose Trap to find five truckloads of dove
    hunters staring dourly at me. I can't imagine why I
    didn't blend right in, being a good old boy from
    southeast NM from way back. Maybe it was the giant
    camera rig, the binoculars, or the shorts and sandals.
    They were blasting away with shotguns right next to the
    trucks, probably to see if they could make me jump (they
    didn't).
  </para>
</day-summary>
```

5.5. The `loc` element: Defining location codes

Each location code used in a day's notes must be defined in a `loc` element. These location codes are used in several places:

- In the `day-loc` attribute of the `day-notes` element.
- In the `default-loc` attribute of the `day-summary` element.

- The `loc`-group pattern within a `form` or `floc` element.

birdnotes.rnc

```
loc = element loc
{
  attribute code { xsd:Name },      1
  attribute name { text },         2
  gps*,                             3
  text?                             4
}
```

- 1 The `code` attribute gives the code being defined. We use the XSchema “ID” datatype because this element defines this location code as a unique identifier so that other attributes can refer to it. It must be a valid XML name, starting with a letter and consisting only of letters, digits, underbar, period, and colon. Example:

```
code='BdA'
```

Originally, this field was defined as a unique identifier of type `xsd:ID`. However, this won't work, because ID values must be unique within a file, and in our case they must be unique only within a `day-notes` element. This problem was caught by the `xmllint` verifier using the Relax NG version of this schema:

```
xmllint --noout --relaxng birdnotes.rng 200705.xml
```

- 2 The `name` attribute gives the ordinary name of that location. For more discussion of the format of the name, see Section 3.2, “The spatial dimension” (p. 4). Example:

```
name='Bosque del Apache NWR: Headquarters'
```

- 3 If there are GPS waypoints relevant to this location, `gps` elements can be included inside the `loc` element; see Section 5.6, “The `gps` element: defining a waypoint” (p. 11).
- 4 Following any `gps` element children, you can include arbitrary text as necessary to describe other relevant information such as directions for getting there, comments on habitat change, local names, etc.

5.6. The `gps` element: defining a waypoint

To note that a given GPS waypoint falls inside a given location, include a `gps` element inside the `loc` element:

birdnotes.rnc

```
gps = element gps
{
  attribute waypoint { text },      1
  text?                             2
}
```

- 1 The `waypoint` attribute is required, and contains the GPS coordinates. For the exact format of these coordinates, see Section 3.2.2, “GPS coordinates” (p. 5). Example:

```
waypoint='340088n 1070839w'
```

- 2 The `gps` element can be empty, or it can include a textual description of that waypoint.

5.7. The day-annotation elements

In the schema, `day-annotation` is an RNC pattern name, not an element name. It enumerates the kinds of things that can follow the `loc` children of the `day-summary` element:

birdnotes.rnc

```
day-annotation =
( para* & route? & weather? & missed? & film? )
```

That is, there can be any number of `para` elements containing general notes; and at most one of each of elements describing the route, the weather, species missed, and film notes. These elements can occur in any order. For the definition of `para`, see Section 5.15, “The narrative elements” (p. 18).

The `route` element allows you to describe what locations were visited and when, to give a picture of the overall sequence of the day's activity. If the route is complex, it can be structured into paragraphs using `para` elements; see Section 5.15, “The narrative elements” (p. 18).

birdnotes.rnc

```
route = element route { narrative }
```

The `weather` element is for general weather conditions.

birdnotes.rnc

```
weather = element weather { narrative }
```

Use `missed` to discuss expected birds that were not seen. For example, it's fairly unusual at any season to bird Bosque del Apache and not see at least one American Coot. This section can also be used to annotate cases where a rarity was pursued but missed.

birdnotes.rnc

```
missed = element missed { narrative }
```

Finally, the `film` element is for recording film exposed on this date, and any other photographic notes.

birdnotes.rnc

```
film = element film { narrative }
```

5.8. The form element: records for one kind of bird

Each `form` element encloses one or more sightings of a given kind of bird. We use “kind” in the taxonomic sense here: sightings attribute to a single species, family, genus, or other taxonomic grouping are all grouped under a single `form` element, even though the sightings may differ in age, sex, or other non-taxonomic differences.

There are two patterns: the `single-sighting` pattern is used for single records, and the `multi-sighting` pattern is used when there are multiple occurrences of the form with different age, sex, location, and so forth.

birdnotes.rnc

```
form = element form
{ taxon-group,      1
  (single-sighting | multi-sighting)
}
single-sighting =
( age-sex-group,    2
  loc-group,        3
  sighting-notes    4
)
```

```
multi-sighting =  
( loc-group,  
  sighting-notes,  
  floc+) 5
```

- 1** The `taxon-group` pattern is a set of three attributes that describe the taxonomic identity of the bird, such as “American Robin” or “hawk sp.” In almost all cases this consists of a single `ab6` attribute whose value is a six-letter bird code. Example for Blue Jay:

```
ab6='blujay'
```

For the general case, including the treatment of hybrids and species pairs, see Section 5.9, “The `taxon-group` pattern: biological classification of the birds seen” (p. 14).

- 2** The `age-sex-group` pattern is a set of several attributes that describe age, sex, and some other aspects of the sighting. See Section 5.10, “The `age-sex-group` attributes” (p. 15).
- 3** The `loc-group` pattern is a set of attributes and other content that specify where the sightings occurred; see Section 5.11, “The `loc-group` pattern: locality attributes and content” (p. 16).
- 4** The `sighting-notes` pattern is a set of optional elements giving additional kinds of information about the sighting such as behavior, description, and such. See Section 5.12, “The `sighting-notes` elements” (p. 16).
- 5** If there are multiple sightings of the same kind of bird, `floc` elements can be included as children of the `form` element. See Section 5.14, “The `floc` element: Multiple sightings of a given form” (p. 17).

For multiple sightings, the parent `form` element can have a `loc-group` specifying the default location for its `floc` children, as well as various `sighting-notes` children that apply to all the sightings of that form.

In the most general case, use a `floc` (“form location”) child element for each different location, age, sex, or other aspect of the different sightings. For example, if the notes include both adult and immature Bald Eagles, the parent `form` element identifies the species (attribute `ab6='baleag'`), and it has two `floc` child elements, one for the adults (attribute `age='a'`) and one for the immatures (`age='i'`), like this:

```
<form ab6='baleag'>  
  <floc age='a' count='2' />  
  <floc age='i' count='1' />  
</form>
```

However, in the great majority of cases, there will be only one sighting of a species in a day's notes. If we always required a `floc` element for information about a sighting, we would need two elements for each sighting. For example, suppose the field notes just say “we saw American Coots today.” If a `floc` element were required, it would have no content, and the XML would look like this:

```
<form ab6='amecoo'>  
  <floc\>  
</form>
```

So, effectively we allow the XML to omit the `floc` child element if there is nothing in it, so the above record becomes much simpler:

```
<form ab6='amecoo' />
```

To reduce redundancy, all locality data not explicitly present in the `form` element are inherited by any child `floc` elements. Here's an example:

```
<form ab6='baleag' loc='BdA' gps='334807.3n 1065305.2w'>
  <loc-detail>
    Perched in the Display Pond.
  </loc-detail>
  <floc age='a' count='2'/>
  <floc age='i' count='1' gps='334803.2n 1065312.3w' />
</form>
```

The first `floc` element will inherit the parent `loc`, `gps` and `loc-detail` values. The second `floc` supplies a different `gps` value, but it inherits the `loc` and `loc-detail` of the parent.

5.9. The taxon-group pattern: biological classification of the birds seen

The taxon-group pattern is a set of three attributes used to classify the bird taxonomically:

```
taxon-group =
(  attribute ab6 { bird-code },           1
  (  attribute rel { "|" | "^" },         2
    attribute alt { bird-code }
  )?,
  attribute notable { "0" | "1" }? 3
)
bird-code = xsd:string { pattern="[a-zA-Z]{2,6}" }
```

birdnotes.rnc

- 1 The `ab6` attribute is required. In most cases it specifies the code corresponding to the name used to describe the classification of the bird. For a discussion of this code system, see Section 3.3, “The taxonomic dimension: what kind of bird?” (p. 6). For example, the code for Wandering Tattler is `wantat`:

```
ab6='wantat'
```

- 2 Normally the `rel` and `alt` attributes are absent, and in such cases the `ab6` attribute describes a bird assigned to a specific species, family, or other taxon.

However, there are two cases where the kind of bird sighted must be described as two six-letter codes (`ab6` and `alt`) and another code (`rel`) that describes the relationship between those codes:

- *Species pairs*: Sometimes the identification can be narrowed down to a choice of two forms, but no further. For example, sightings of certain flycatchers in the genus *Empidonax* are often described as “Dusky/Hammond’s Flycatcher.” In these cases we use `rel='|'`, for example:

```
ab6='dusfly' rel='|' alt='hamfly'
```

- *Hybrids*: The classification is given as a pair of codes with `rel='^'`. For example, a hybrid of Mallard x Gadwall would be encoded:

```
ab6='gadwal' rel='^' alt='mallar'
```

- 3 Flag unusual records with `notable='1'`. Such records should be highlighted so that readers not interested in routine sightings can more rapidly skim the reports for “the good stuff.”

For species pairs and hybrids, the order is not important. However, retrieval should always ensure that the `ab6` code is lexically less than the `alt` code, so that records of a given pair or hybrid fall in one place and not two.

5.10. The age-sex-group attributes

birdnotes.rnc

```
age-sex-group =  
(  
  attribute age { "u" | "a" | "i" | "p" }?, 1  
  attribute sex { "u" | "m" | "f" }?, 2  
  attribute q { " " | "?" | "-" }?, 3  
  attribute count { text }?, 4  
  attribute fide { text }? 5  
)
```

- 1** Age class of the birds seen:

u	Unknown age; the default.
a	Adult.
i	Immature, subadult, or juvenal.
p	Female or immature (ϕ), which is actually a combination of an age constraint and a sex constraint: it's not an adult male.

- 2** Sex of the birds seen. In some of the author's data sets, code "p" can occur, because this stands for "female or immature" and is thus both an age and a sex. However, to simplify processing, we will force that value to occur only in age codes.

u	Unknown (the default).
m	Male.
f	Female.

- 3** If the observer is not certain of the identification, use attribute `q='?'` to flag the record as questionable. Such records should be considered a best guess, but not useable for scientific purposes, or countable in a birding context (such as for the purposes of reporting totals to the American Birding Association).

Code `q='-'` indicates that the identity is not in question, but it is not ABA countable.

- 4** For many records, no count of individuals is given. However, the `count` attribute is available for specifying counts or estimates in various ways:

- Actual numbers or estimates such as `count='3'` or `count='50000'`.
- `count='#'` means "more than one", and is the default interpretation.
- A "+" suffix means "or more;" for example, `count='9+'` means nine or more.
- A "-" suffix means "or less;" such as `count='6-'` for "not more than six."
- If the estimate is a range, use two numbers with a hyphen between them. For example, `count='30-40'` means there were about thirty to forty individuals.
- A tilde (~) prefix is used to indicate an approximate count, e.g., "~100" for "about a hundred."

- 5** When the primary observer did not personally see the bird, but wishes to pass on secondhand reports, use a `fide` attribute whose value is the name or names of the actual observers. For example:

```
fide='Glenn Kendall'
```

One should assume that the veracity of the record is that of the stated observer. Such records are not considered countable for birding totals.

5.11. The `loc-group` pattern: locality attributes and content

The `loc-group` pattern is a combination of two attributes and a child element. These three items specify the location of the sighting.

birdnotes.rnc

```
loc-group =
(
  attribute loc { xsd:Name }?,      1
  attribute gps { text }?,         2
  loc-detail?                       3
)
loc-detail = element loc-detail { narrative }
```

- 1 If the sightings all occurred within an area defined by a location code, this attribute is a reference to the definition of that code in a `loc` element within the day's `day-summary` element. If a sighting has no explicit `loc` attribute, its value defaults to the code given in the `default-loc` attribute of the day's `day-summary` element.

For example, a sighting of a Merlin at Bosque del Apache (location code `BdA`):

```
<form ab6='merlin' count='1' loc='BdA' />
```

- 2 You can attach a GPS waypoint's coordinates to this sighting with this attribute. Example:

```
<form ab6='heptan' gps='340058n 1070839w' />
```

For the general form of this attribute, see Section 3.2.2, "GPS coordinates" (p. 5).

- 3 You can add arbitrary text describing the details of the sighting's locality by enclosing it in a `loc-detail` element. Example of a conspicuous Greater Roadrunner on a golf course:

```
<form ab6='greroa' count='1' loc='GC'>
  <loc-detail>
    Perched on the 4th tee marker monument stone.
  </loc-detail>
</form>
```

For the definition of `narrative`, see Section 5.15, "The narrative elements" (p. 18).

5.12. The `sighting-notes` elements

The `sighting-notes` pattern is a set of optional child elements that can occur inside a `form` or `floc` element:

birdnotes.rnc

```
sighting-notes = ( desc? & behavior? & voc? & breeding? &
  photo* & para* )
desc = element desc { narrative }
behavior = element behavior { narrative }
voc = element voc { narrative }
breeding = element breeding { narrative }
```

Most of these child elements are simply containers for narrative; see Section 5.15, “The narrative elements” (p. 18).

desc

Plumage and other physical descriptions: feather colors and patterns, soft part colors, shape, and other static appearance details.

behavior

Notes on behavior other than vocalization.

voc

Vocalizations.

breeding

Evidence of breeding—solid or circumstantial, attempted or successful. Courtship behavior, nest-building, eggs, young, fledging, and any other breeding behavior that implies that the locality of the sighting is the locality of breeding. Exclude sightings of young that are capable of independent locomotion.

photo

Each `photo` element describes photographic evidence of the sighting. For the structure of this element, see Section 5.13, “The `photo` element” (p. 17).

para

For any remarks about the sighting that don't fit into the other categories.

5.13. The `photo` element

The purpose of the `photo` element is to link the sighting with the observer's library of photographs.

birdnotes.rnc

```
photo = element photo
{
  attribute cat-no { text },           1
  attribute url { xsd:anyURI }?,     2
  text?                               3
}
```

- 1** The `cat-no` attribute is required, and specifies the catalog number of the image.
- 2** If there is a rendering of this photo available on the web, use the `url` to specify its Universal Resource Identifier. Optional.
- 3** You can add text content to a `photo` element for remarks on the photo.

5.14. The `floc` element: Multiple sightings of a given form

When there are multiple sightings of one kind of bird, but the locations or age or sex or other details are different, you can place each sighting inside the `form` element enclosed in its own `floc` element. The intent is that there be only one `form` element for each kind of bird in a day's notes.

The structure of a `floc` element is very similar to the structure of the `form` element, but it lacks the `taxon-group` that describes the taxonomic identity:

birdnotes.rnc

```
floc = element floc { single-sighting }
```

The content of a `floc` element is described above; see Section 5.8, “The `form` element: records for one kind of bird” (p. 12).

Here's an example of the use of `floc` elements. If we have a single record of one American Robin (code `amerob`), it might look like this:

```
<form ab6='amerob' count='1' />
```

But if we saw three adults and five young:

```
<form ab6='amerob'>
  <floc count='3' age='a' />
  <floc count='5' age='i' />
</form>
```

If no location is given in the `floc` element, it inherits the location from the containing `form` element, or from the `default-loc` attribute of the `day-summary` element if the `form` element gives no location code.

5.15. The narrative elements

In quite a number of different places, we want to be able to represent general narratives—descriptions of places, birds, weather, and such. In these narrative sections, we want to be able to use special inline tags to markup elements that need special typography. In particular, we want to present scientific names in italics.

Another important structural feature of narrative content is the ability to break it up into paragraphs. For example, in the route description, it is useful to be able to break up the major sections of the routes into separate paragraphs, rather than presenting it as all one huge glob.

We could use something like DocBook's `para` element to hold such narrative, and use a content model something like “`para+`” to allow one or more paragraphs.

However, most of the time, narrative content will be rather short, and it is somewhat burdensome to force the writer to wrap just a short note inside a `para` element.

The solution in these locations is to allow either a sequence of `para` elements, or just the content that goes inside a `para`. In the latter case, there is an implied `para` element wrapping up the content.

Here are the relevant parts of the schema for narrative content. First, we define the `para` element itself:

```
para = element para
{ para-content
}
para-content = mixed { para-markup* }
para-markup = (genus|cite)
genus = element genus { text }
cite = element cite { text }
```

birdnotes.rnc

para-content

This pattern describes what can go inside a paragraph. The Relax NG “`mixed`” construct means that the content may be any mixture of plain text, or occurrences of either `genus` or `cite` inline elements.

para-markup

This pattern enumerates the elements that can occur inside an explicit or implied `para` paragraph.

genus

An inline element containing a scientific name of genus rank or lower, to be rendered in italics.

cite

An inline element containing a citation of a book, article, movie, or other reference whose name should be rendered in italics.

Finally, we define the `narrative` pattern: either a sequence of paragraphs, or mixed content that can occur inside a paragraph.

birdnotes.rnc

```
narrative = ( para+ | para-content )
```

6. Directory structure for data files

Here is a suggested structure for managing data files using the schema described in Section 5, “The schema” (p. 7). This structure provides a reasonably modular approach to managing the data for one observer.

1. Pick a root directory for your data files.
2. For each year when field notes were taken, create a subdirectory whose name is the four-digit year.
3. For each month, store the notes for that month in chronological order in a file whose name has the form `YYYY/YYYY-MM.xml`, where `YYYY` is the four-digit year number and `MM` is the two-digit month number.

For example, notes from January 2010 would be kept in file “2010/2010-01.xml”.

The advantage to structuring your data files in this way is that they can be managed by a single Python-language object, described in Section 21, “class `BirdNoteTree`: A complete set of notes” (p. 30).

7. `birdnotes.py`: A Python interface

To help processing of files using the schema described above, we provide a Python module that reads and writes XML. To use this module, you must have these files in the current directory:

- `birdnotes.py`¹¹: The principal Python module.
- `rnc.py`¹²: A module containing symbolic names for all the element and attribute names from the schema.
- All the program and data files required for the taxonomy package, *A system for representing bird taxonomy*¹³: `txny.py`, `rnc_txny.py`, and one of the AOU Check-List files, named `aou.xml`.

Once these files are installed, your Python script must import it with a line like this:

```
from birdnotes import *
```

Here is the general flow of a program that reads notes using this module:

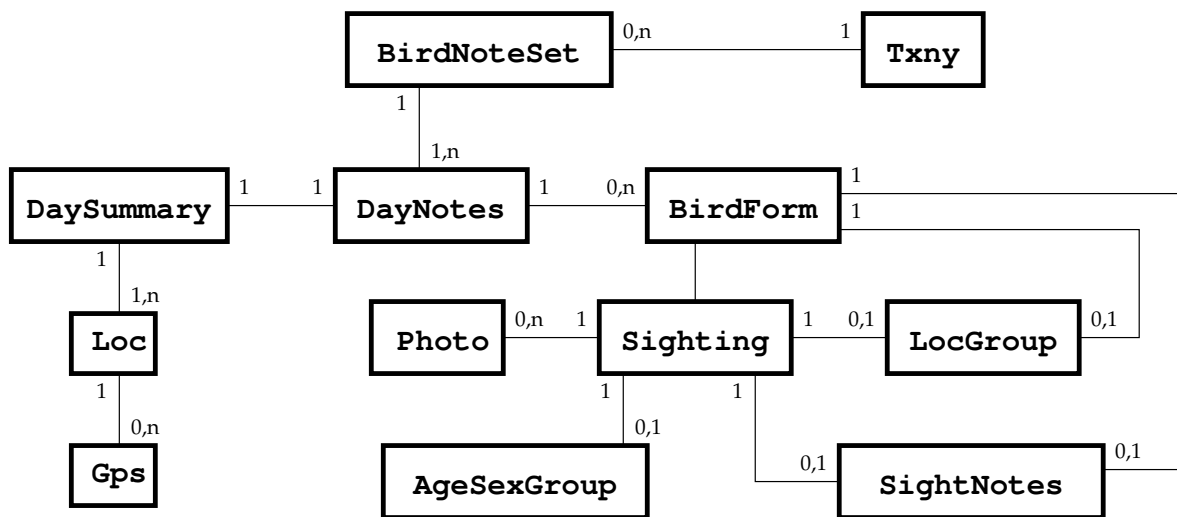
1. Instantiate a `Txny` object representing the `aou.xml` file defining the desired taxonomic arrangement of birds.
2. Instantiate a `BirdNoteSet` object, passing it the `Txny` instance.
3. For each XML file you want to read, call the `.readFile()` method on the `BirdNoteSet` object.

¹¹ <http://www.nmt.edu/~shipman/aba/doc//pyims/birdnotes.py>

¹² <http://www.nmt.edu/~shipman/aba/doc//pyims/rnc.py>

¹³ <http://www.nmt.edu/~shipman/xnomo/>

The interfaces to the `BirdNoteSet` class and the various other classes are defined in separate sections below. Here is a diagram showing the classes and their relationships, except for `Txny` which is external to this module.



- Section 8, “class `BirdNoteSet`: Container for notes” (p. 20).
- Section 9, “class `DayNotes`: Notes for one day” (p. 21).
- Section 10, “class `DaySummary`: Daily context” (p. 23).
- Section 11, “class `Loc`: Locality code definition” (p. 24).
- Section 12, “class `Gps`: GPS waypoint” (p. 24).
- Section 13, “class `BirdForm`: Records for one kind of bird” (p. 25).
- Section 14, “class `LocGroup`: Inheritable locality data” (p. 26).
- Section 15, “class `SightNotes`: Sighting notes” (p. 26).
- Section 16, “class `Sighting`: Single sighting” (p. 27).
- Section 17, “class `AgeSexGroup`: Age and sex details” (p. 28).
- Section 18, “class `Photo`: Photograph link” (p. 28).

In addition to the classes shown above, two classes are used throughout the structure to represent mixed content as defined in Section 5.15, “The narrative elements” (p. 18).

- Section 19, “class `Narrative`: General container for narrative” (p. 28).
- Section 20, “class `Paragraph`: General container for a paragraph of narrative” (p. 29).

8. class `BirdNoteSet`: Container for notes

An instance of this class holds one or more daily note sets.

8.1. `BirdNoteSet()`: Constructor

To create a `BirdNoteSet` object, you will need a `Txny` instance T that defines your preferred taxonomic arrangement:

```
BirdNoteSet(T)
```

Attributes of the instance include:

.txny

As passed to the constructor.

.period

The period title, having the form “*monthName year*”. Initially an empty string. After you read an XML file into the instance with the `.readFile()` method, this attribute will contain the period name from that set.

8.2. `BirdNoteSet.genDays()`: Generate contained daily sets

This method generates a sequence of the contained `DayNotes` instances. It takes no arguments. The generated instances may not be in any predictable order. See Section 9, “`class DayNotes`: Notes for one day” (p. 21).

There may be multiple `DayNotes` instances for the same date and state. For example, when performing censuses such as the Christmas Bird Count and North American Migration Count, the author prefers to keep census records in one `day-notes` element, separating into another `day-notes` element casual observations not made within the census methodology.

8.3. `BirdNoteSet.addDay()`: Add a daily note set

To add a `DayNotes` instance *D* to a `BirdNoteSet` instance *B*:

```
B.addDay(D)
```

8.4. `BirdNoteSet.readFile()`: Read an XML file

This method on a `BirdNoteSet` instance *B* reads a set of bird notes from a file *fileName*:

```
B.readFile(fileName)
```

If there are any problems reading the file, this constructor will raise an `IOError` exception.

8.5. `BirdNoteSet.writeFile()`: Save as XML

To write all the records contained in a `BirdNoteSet` instance *B* to a file named *fileName*:

```
B.writeFile(fileName)
```

9. `class DayNotes`: Notes for one day

An instance of this class is a container for all the notes for one day within one state.

Attributes include:

.noteSet

The parent `BirdNoteSet` instance that contains this instance.

.regionCode

The lowercased postal code, e.g., 'nm' for New Mexico.

.date

The date as a string, formatted as *YYYY-MM-DD*.

.daySummary

A `DaySummary` instance representing the day's general features. See Section 10, “class `DaySummary`: Daily context” (p. 23).

.dayLoc

A locality that includes all the day's notes, as a `LOC` instance. See Section 11, “class `LOC`: Locality code definition” (p. 24).

9.1. `DayNotes.title()`: General title string

This method returns a string of the form

```
yyyy-mm-dd rr: dayloc
```

where *yyyy-mm-dd* is the date, *rr* is the region code (uppercased), and *dayloc* is the full name corresponding to the `.dayLoc` attribute.

Here's an example return value:

```
2007-09-05 NM: Hobbs
```

9.2. `DayNotes.defaultLoc()`: What is the default location?

This method takes no arguments and returns the day's default locality as a `LOC` instance; see Section 11, “class `LOC`: Locality code definition” (p. 24).

9.3. `DayNotes.lookupLoc()`: Look up a location code

To find the definition of a locality code *code*, given a `DayNotes` instance *D*:

```
D.lookupLoc(code)
```

If the code is defined, the method returns its definition as a `LOC` instance; see Section 11, “class `LOC`: Locality code definition” (p. 24).

If the code is not defined, the method raises a `KeyError` exception.

9.4. `DayNotes.addForm()`: Add a new form record

To add one or more sightings represented by a `BirdForm` instance *B* to a `DayNotes` instance *D*:

```
D.addForm(B)
```

See Section 13, “class `BirdForm`: Records for one kind of bird” (p. 25).

9.5. `DayNotes.genForms()`: Retrieve stored sightings

To generate all the sightings contained in a `DayNotes` instance *D*, call its `.genForms()` method with no arguments. The method will generate a sequence of `BirdForm` instances representing the contained sightings. See Section 13, “class `BirdForm`: Records for one kind of bird” (p. 25).

9.6. DayNotes.genFormsSeq(): Retrieve sightings in addition order

As sightings are added to a DayNotes instance *D*, a record is kept of the order in which they were added. To retrieve these sightings in the same order, use the method *D.genFormsSeq()* with no arguments. The method will generate the contained sightings as a sequence of BirdForm instances.

10. class DaySummary: Daily context

An instance of this class is a container for various information that applies to all the sightings for a given DayNotes instance.

Public attributes include:

.route

A description of the route as an instance of Section 19, “class Narrative: General container for narrative” (p. 28), or None if there is no route description.

.weather

A description of the weather as a Narrative instance, or None if there is no weather narrative.

.missed

Discussion of missed species as a Narrative instance, or None.

.film

Photographic notes as a Narrative instance, or None.

.notes

General notes as a Narrative instance, or None.

10.1. DaySummary.defaultLoc(): What is the default location?

For a DaySummary instance *D*, the method *D.defaultLoc()* takes no arguments and returns a Loc instance representing the default location for this day-notes element. See Section 11, “class Loc: Locality code definition” (p. 24).

10.2. DaySummary.addLoc(): Add a new locality code definition

To add a new locality definition *L* (as an instance of Section 11, “class Loc: Locality code definition” (p. 24)) to a DaySummary instance *D*, use this method call:

```
D.addLoc(L)
```

10.3. DaySummary.lookupLoc(): Look up a location code

Given a DaySummary instance *D*, to look up a location code string *s*, use this method call:

```
D.lookupLoc(s)
```

If the code is defined, the method returns its definition as a Loc instance; see Section 11, “class Loc: Locality code definition” (p. 24). If the code is undefined, it raises a KeyError exception.

10.4. DaySummary.genLocs(): Retrieve all locality definitions

For a DaySummary instance D , this method call generates all the locality definitions in D as a sequence of LOC instances; see Section 11, “class LOC: Locality code definition” (p. 24).

```
D.genLocs()
```

11. class Loc: Locality code definition

An instance of this class represents the definition of a locality code used in a specific day - notes element. Public attributes include:

.code

The locality code as a string.

.name

The full name of the locality as a string.

.text

The full description of the locality as a string, or None if there is no long description.

11.1. Loc.addGps(): Add a waypoint

To add a waypoint definition W (as an instance of Section 12, “class Gps: GPS waypoint” (p. 24)) to a LOC instance L :

```
L.addGps(W)
```

11.2. Loc.genGps(): Generate waypoints

To retrieve all the GPS waypoints stored in a LOC instance L , use this method:

```
L.genGps()
```

The method generates the waypoints as a sequence of instances of Section 12, “class Gps: GPS waypoint” (p. 24).

12. class Gps: GPS waypoint

Each instance of this class represents one waypoint from a Global Position System receiver. Attributes include:

.waypoint

The lat-long coordinates as a string. For the format, see Section 5.6, “The gps element: defining a waypoint” (p. 11).

.text

Textual description of the locality, or None if there is no description.

.latLon

A `LatLon` instance representing this waypoint. For documentation on this class, see *A Python mapping package*¹⁴.

13. class BirdForm: Records for one kind of bird

Each instance of this class represents a group of one or more sightings of the same kind of bird. An instance is a container for one or more `Sighting` instances.

Public attributes:

.dayNotes

The parent `DayNotes` instance.

.birdId

An instance of class `BirdId` that describes what kind of bird was observed. For documentation, see the documentation for `abbr.py` in *A system for representing bird taxonomy*¹⁵.

.notable

True iff this form at this time and place is out of the ordinary.

.locGroup

For the single sighting case, this attribute is `None`. If there is any `loc-group` content under the input `form` element, it is in the sole child `Sighting`.

For the multiple sighting case, if there is any `loc-group` content under the `form` element that is not inside one of the child `floc` elements, it is attached here as a `LocGroup` instance. See Section 14, “class `LocGroup`: Inheritable locality data” (p. 26).

.sightNotes

For the single sighting case, this attribute is `None`. If there is any `sighting-notes` content, it resides in the sole child `Sighting`.

For the multiple sighting case, if there are any `sighting-notes` content under the `form` element that is not inside one of the child `floc` elements, it is attached here as a `SightNotes` instance. See Section 15, “class `SightNotes`: Sighting notes” (p. 26).

13.1. BirdForm.addSighting()

To add a `Sighting` instance *S* to a `BirdForm` instance *B*:

```
B.addSighting(S)
```

13.2. BirdForm.__len__(): Return the number of sightings

For a `BirdForm` instance *B*, this expression returns the number of sightings of this form:

```
len(B)
```

¹⁴ <http://infohost.nmt.edu/tcc/help/lang/python/mapping/doc/>

¹⁵ <http://www.nmt.edu/~shipman/xnomo/ifc-abbr.html>

13.3. BirdForm.__getitem__(): Retrieve a child sighting

For a BirdForm instance *B*, this expression returns the *N*th child of *B* (counting from 0) as a Sighting instance.

```
B[N]
```

13.4. BirdForm.nSightings(): Number of sightings

For a BirdForm instance *B*, method call *B.nSightings()* returns the number of Sighting instances contained within *B*.

13.5. BirdForm.getLocGroup(): Get the effective locality

For a BirdForm instance *B*, this method call returns information about the sighting locality as an instance of Section 14, “class LocGroup: Inheritable locality data” (p. 26):

```
B.getLocGroup()
```

14. class LocGroup: Inheritable locality data

There are three kinds of locality data that can be attached to records in this system:

- A locality code, as defined in a `loc` element.
- A GPS waypoint as a lat-lon string.
- A general description of the location from a `loc-detail` element.

An instance of `LocGroup` represents any combination of these three.

Public attributes of a `LocGroup` instance include:

.loc

If this group is referred to specific locality code, this attribute contains the definition of the code as an instance of Section 11, “class Loc: Locality code definition” (p. 24), otherwise `None`.

.gps

If the group is referred to a waypoint, this attribute contains the lat-lon as a string, in the same format as the `waypoint` attribute defined in Section 12, “class Gps: GPS waypoint” (p. 24), otherwise `None`.

.locDetail

If there is a narrative description of the locality, this attribute contains that description as an instance of Section 19, “class Narrative: General container for narrative” (p. 28), else `None`.

15. class SightingNotes: Sighting notes

An instance of this class holds the content defined in Section 5.12, “The sighting-notes elements” (p. 16). Public attributes:

.desc

An instance of Section 19, “class Narrative: General container for narrative” (p. 28) containing a description of the bird’s appearance, or `None`.

.behavior

Behavior notes as a `Narrative` instance, or `None`.

.voc

Vocalization notes as a `Narrative` instance, or `None`.

.breeding

Breeding notes as a `Narrative` instance, or `None`.

.notes

General remarks as a `Narrative` instance, or `None`.

15.1. `SightNotes.addPara()`: Add general notes

To add a paragraph of general remarks to a `SightNotes` instance *S*, package the remarks in an instance *P* of Section 20, “`class Paragraph`: General container for a paragraph of narrative” (p. 29) and call this method:

```
S.addPara(P)
```

15.2. `SightNotes.addPhoto()`: Add a photo link

To add a photo link as a `Photo` instance *P* to a `SightNotes` instance *S*:

```
S.addPhoto(P)
```

15.3. `SightNotes.genPhotos()`: Retrieve photo links

For a `SightNotes` instance *S*, this method call generates all the contained photo links as a sequence of instances of Section 18, “`class Photo`: Photograph link” (p. 28):

```
S.genPhotos()
```

16. `class Sighting`: Single sighting

An instance of this class represents a sighting of one or more birds that share locality, age, sex, or other details. Public attributes:

.birdForm

The kind of bird as an instance of Section 13, “`class BirdForm`: Records for one kind of bird” (p. 25).

.locGroup

If the instance has any locality information attached to it, this attribute contains that information as an instance of Section 14, “`class LocGroup`: Inheritable locality data” (p. 26).

.ageSexGroup

If the instance specifies any of the content described in Section 5.10, “The age-sex-group attributes” (p. 15), this attribute will be an instance of Section 17, “`class AgeSexGroup`: Age and sex details” (p. 28), otherwise `None`.

.sightNotes

If the instance has any of the sighting data described in Section 5.12, “The sighting-notes elements” (p. 16), this attribute will be an instance of Section 15, “class SightNotes: Sighting notes” (p. 26), otherwise None.

16.1. Sighting.getLocGroup()

To retrieve the locality information of a `Sighting` instance `S`:

```
S.getLocGroup()
```

The result is returned as an instance of Section 14, “class LocGroup: Inheritable locality data” (p. 26).

17. class AgeSexGroup: Age and sex details

An instance of this class is a container for information as defined in Section 5.10, “The age-sex-group attributes” (p. 15); refer to that section for details about the possible values. Public attributes:

.age

An age code or None.

.sex

A sex code or None.

.q

A countability code or None.

.count

A string describing the number of individuals or None.

.fide

A string naming the actual observer, or None.

18. class Photo: Photograph link

An instance of this class represents a link to a photograph. Public attributes:

.catNo

The photo's standard catalog number.

.url

The URL where this photo can be viewed, or None.

.text

Comment text as a string, or None.

19. class Narrative: General container for narrative

An instance of this class is a container for one or more paragraphs of text. The constructor has this calling sequence:

```
Narrative()
```

It returns a new, empty instance.

19.1. `Narrative.addPara()`: Add a paragraph

To add a paragraph to a `Narrative` instance N , wrap the paragraph in an instance P of Section 20, “`class Paragraph`: General container for a paragraph of narrative” (p. 29), and call this method:

```
N.addPara(P)
```

19.2. `Narrative.__len__()`: How many paragraphs?

For a `Narrative` instance N :

```
len(N)
```

returns the number of paragraphs.

19.3. `Narrative.__getitem__()`: Get one paragraph

For a `Narrative` instance N , this function returns the K th paragraph:

```
N[K]
```

19.4. `Narrative.genParas()`: Generate the contained paragraphs

To extract the paragraphs from a `Narrative` instance N , use this method:

```
N.genParas()
```

It generates the content of N as a sequence of `Paragraph` instances.

20. `class Paragraph`: General container for a paragraph of narrative

An instance of this class is a container for mixed content as defined in Section 5.15, “The narrative elements” (p. 18). The constructor takes this form, and returns a new, empty instance:

```
Paragraph()
```

20.1. `Paragraph.addContent()`: Add content

To add content to a `Paragraph` instance P , use this method:

```
P.addContent(tag, text)
```

tag

To add ordinary text to the paragraph, pass `None` as this argument. To add text marked up with one of the element names in the `para-markup` pattern of the schema, pass the element name as this argument.

text

A string containing the text to be added.

Here is an example. Suppose you want to represent this text:

Read about *Tyrannus couchii* in *NMOS Journal* next month.

This code sequence would build a `Paragraph` instance `p` with that content:

```
p = Paragraph()  
p.addContent ( None, 'Read about ' )  
p.addContent ( 'genus', 'Tyrannus couchii' )  
p.addContent ( None, ' in ' )  
p.addContent ( 'cite', 'NMOS Journal' )  
p.addContent ( None, ' next month.' )
```

20.2. Paragraph.genContent(): Generate the phrase list

This method generates the content of a `Paragraph` instance `P`:

```
P.genContent()
```

The method generates a sequence of tuples (`tag`, `text`), just as those tuples were added with the `.addContent()` method.

There is no guarantee about the sequence of tag values. Unmarked text may follow unmarked text, and the tagged sections may occur anywhere.

21. class BirdNoteTree: A complete set of notes

If the notes are stored in a directory tree, divided into yearly subdirectories with monthly files as described in Section 6, “Directory structure for data files” (p. 19), you can use a single object to manage a complete set of notes files.

The intent of the `BirdNoteTree` class is to extract records from such a directory tree. In particular, it is intended for a Web-based query interface, where the user may be interested only in records from specific regions or periods.

To save time and computer memory, this class does not immediately read every notes file in the tree. Once you have instantiated the class, you will use its `.genMonths()` method to generate a sequence of `BirdNoteSet` instances, one per month.

The `.genMonths()` method includes options for selecting records by date and by season. For example, if the user is interested only in summer records from a specific decade, the program reads only those monthly files that are from summer months in those years.

21.1. BirdNoteTree(): Constructor

To open a directory tree, use this calling sequence:

```
BirdNoteTree ( T, rootDir="." )
```

The `T` argument is an instance of class `Txny` (as used in Section 8.1, “`BirdNoteSet(): Constructor`” (p. 20)).

The optional *rootDir* argument is the path name of the root directory. The default is “.”, the current working directory.

Attributes of the instance include:

.txny

As passed to the constructor.

.rootDir

As passed to the constructor.

21.2. BirdNoteTree.genMonths(): Generate monthly record sets

To extract records from a tree represented by an instance *B* of the `BirdNoteTree` class, use this method:

```
B.genMonths ( startDate=None, endDate=None, startSeason=None,
              endSeason=None )
```

This method generates a sequence of zero or more `BirdNoteSet` instances, each containing the entire set of records for one month file in the directory structure. The optional arguments allow you to extract only months from a specific time period.

startDate

To select only months after a given date, pass to this argument an instance of the `date` class of the standard Python `datetime` package. The method will generate `BirdNoteSet` instances for only the month containing that date and later months.

endDate

To ignore months after a given date, pass a `datetime.date` instance to this argument giving the last date of interest.

startSeason

To ignore months after a given date in *all* years, pass to this argument a `datetime.date` instance whose month and day specify the first date of interest. The year value in this argument will be ignored.

endSeason

To ignore months after a given date in all years, pass a `datetime.date` instance whose month and day specify the last date of interest. The year value will be ignored.

Note

The arguments provide filtering of records only down to the month level. The generated `BirdNoteSet` instances will still contain all the data from the original files. This level of filtering is provided only to speed up the retrieval of records by skipping files totally outside the dates and seasons of interest.

Here is a skeletal example. Suppose you want to extract all records, and further suppose that the current working directory is the root directory for the data files and that the `ao.xml` file containing the desired taxonomy is also in the current directory. The code would look something like this:

```
from txny import *
from birdnotes import *
import datetime
...
txny = Txny()
```

```
tree = BirdNoteTree(txny)
for monthSet in tree.genMonths():
    process monthSet, a BirdNoteSet instance
```

Suppose that you want only months that contain only records from September 23, 1997, through February 10, 1998, inclusive. Replace the last two lines of the above example with these lines:

```
start = datetime.date ( 1997, 9, 23 )
end = datetime.date ( 1998, 2, 10 )

for monthSet in tree.genMonths(start, end):
    process monthSet, a BirdNoteSet instance
```

And for a final example, to extract only records from May 5 through May 20:

```
start = datetime.date ( 2000, 5, 5 )
end = datetime.date ( 2000, 5, 20 )
for monthSet in tree.genMonths(seasonStart=start, seasonEnd=end):
    process monthSet, a BirdNoteSet instance
```

In the example above, the year will be ignored, and the `.genMonths()` method will generate one `BirdNoteSet` instance for every year that had a file for May.

22. class FlatSighting: Complete sighting record

External consumers of bird sighting data often represent sightings as spreadsheets or database tables. To make it easier to export data into those tools, the `FlatSighting` class centralizes the most important facts about a bird sighting into a single entity.

The constructor operates on a `Sighting` instance *S*:

```
FlatSighting ( S )
```

Attributes of this instance:

.txKey

Taxonomic key number for this kind of bird, as described in the definition of the `Taxon` class¹⁶.

.abbr

First or only bird code, stripped of trailing blanks. This code, the one in the `.abbr2` attribute, and the `.rel` field are documented in *A system for representing bird taxonomy*¹⁷.

.rel

If this is a single-code form, contains the empty string. If this is a compound form, contains " | " for species pairs, or "^" for hybrids.

.abbr2

If this is a single-code form, contains the empty string. If this is a compound form, contains the second bird code, stripped of trailing blanks.

.eng

The English name in "Generic[, Specific]" form. Examples: "Hawk, Hawaiian"; "Gadwall x Shoveler, Northern".

¹⁶ <http://www.nmt.edu/~shipman/xnomo/ifc-Taxon.html>

¹⁷ <http://www.nmt.edu/~shipman/xnomo/ifc-abbr.html>

.age

The age code, or an empty string if unknown.

.sex

The sex code, or an empty string if unknown.

.q

The countability flag: an empty string normally; "-" for ABA uncountable; "?" for unsure ID.

.count

The count field. Note that this is not necessarily a single number! Examples: "1"; "#" for two or more; "200-250", a range; "10+", an open-ended range. For an unknown count, it holds an empty string, which is to be interpreted as "one or more."

.date

The date as "YYYY-MM-DD".

.regionCode

Two-character U.S. state code or other region code, capitalized. Always present.

.locName

Brief description of the general locality.

.observer

If the primary observer did not see this bird, the name of the observer who did; otherwise an empty string.

This method is available:

.delimited(delimiter='\t'):

Return the record as a string, with the attributes in the order shown above (.txKey through .observer), with the fields separated by the supplied *delimiter*.

