

# noteweb: Rendering bird field notes for the Web

Zoological  
Data Processing

John W. Shipman

2011-07-04 13:28

## Abstract

Describes a system for translating XML-based bird field notes to HTML.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to [john@nmt.edu](mailto:john@nmt.edu).

## Table of Contents

1. Introduction .....	4
2. Operation .....	4
2.1. Further work .....	4
3. Overview of the internals .....	4
4. The generated XHTML .....	5
4.1. XHTML for the index page .....	5
4.2. XHTML for the month page .....	6
4.3. XHTML for one day-notes .....	7
4.4. XHTML for the day-summary block .....	8
4.5. XHTML for locality definitions .....	8
4.6. XHTML for day-annotation elements .....	9
4.7. XHTML rendering of narrative elements .....	9
4.8. XHTML for the form element .....	9
4.9. XHTML for bird form names .....	10
4.10. XHTML rendering of form data .....	10
4.11. XHTML rendering of sighting notes .....	11
4.12. XHTML rendering of photo links .....	12
4.13. XHTML rendering of floc elements .....	12
5. The style sheet .....	12
5.1. General style .....	12
5.2. Inline markup rules .....	12
5.3. table.seasons .....	13
5.4. Seasonal column markup rules: th.winter, etc. ....	13
5.5. CSS rules for rows of the index table .....	14
5.6. th.row-label: The row label .....	14
5.7. div.loc-child: Indented child block .....	14
5.8. div.day-summary: Daily summary block .....	15
5.9. div.loc-def: Location definition .....	15
5.10. div.loc-narrative: Locality narrative .....	15
5.11. div.para: Ordinary paragraphs .....	15

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

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

5.12. <code>div.form</code> : General form-related data .....	16
5.13. <code>div.floc</code> : Multiple sightings .....	16
6. Design notes .....	16
6.1. Discarded approaches .....	17
7. Overall program flow .....	18
8. Prologue .....	18
9. Imported modules .....	18
10. Manifest constants .....	19
10.1. <code>MONTH_NAME_MAP</code> : Translate month numbers to month names .....	19
10.2. <code>MONTH_SEASON_MAP</code> : Define the season for each month .....	19
10.3. <code>YEAR_PAT</code> : Pattern for year numbers .....	20
10.4. <code>YYYY_MM_XML_PAT</code> : Month file name pattern .....	20
10.5. <code>HTML_EXT</code> : File extension for XHTML pages .....	20
10.6. <code>INDEX_PAGE_NAME</code> : Name of the index page .....	20
10.7. <code>INDEX_PAGE_TITLE</code> .....	20
10.8. <code>HOME_PAGE_URL</code> : Home page URL .....	21
10.9. <code>CONVENTIONS_URL</code> .....	21
10.10. <code>SEASONS_CLASS</code> : Class attribute for the table of seasons .....	21
10.11. <code>YEAR_GROUP_FREQUENCY</code> .....	21
10.12. <code>YEAR_GROUP_CLASS</code> : CSS class for years ending a year group .....	21
10.13. <code>YEAR_ROW_CLASS</code> .....	21
10.14. <code>ROW_LABEL_CLASS</code> .....	21
10.15. <code>NBSP</code> : Non-breaking space character .....	22
10.16. <code>PHI</code> : Greek letter $\phi$ .....	22
10.17. <code>CSS_URL</code> : Our stylesheet .....	22
10.18. <code>ZDP_LOGO</code> .....	22
10.19. <code>ZDP_URL</code> .....	22
10.20. <code>LOC_CHILD_CLASS</code> .....	22
10.21. <code>NOTABLE_CLASS</code> .....	22
10.22. <code>DAY_SUMMARY_CLASS</code> .....	23
10.23. <code>LOC_DEF_CLASS</code> .....	23
10.24. <code>LOC_NARRATIVE_CLASS</code> .....	23
10.25. <code>LOC_LABEL_CLASS</code> .....	23
10.26. <code>PARA_CLASS</code> .....	23
10.27. <code>FORM_CLASS</code> .....	23
10.28. <code>NOTABLE_FORM_CLASS</code> .....	23
10.29. <code>BIRD_NAME_CLASS</code> .....	24
10.30. <code>FLOC_CLASS</code> .....	24
10.31. <code>GENUS_CLASS</code> .....	24
11. <code>main()</code> : The main program .....	24
12. <code>findYears</code> : Locate year directories .....	26
13. <code>buildIndex()</code> : Build the index page .....	27
14. <code>pageFrame</code> : Set up navigation for the index page .....	28
15. <code>indexBoilerplate</code> : Fixed content for the index page .....	29
16. <code>indexTable()</code> : Build the table of monthly links .....	29
17. <code>indexTableFrame()</code> : Set up the table structure .....	30
18. <code>buildRow()</code> : Build one row of the index table .....	31
19. <code>buildMonthCell()</code> : Build one monthly cell in the index table .....	33
20. <code>class Args()</code> : Command line argument processor .....	34
21. <code>class YearCollection</code> : The top-level data structure .....	35
21.1. <code>YearCollection.__init__()</code> : Constructor .....	36
21.2. <code>YearCollection.addYear()</code> : Add a year .....	36

21.3.	YearCollection.__getitem__(): Return self[yyyy]	37
21.4.	YearCollection.genYearsRev(): Generate years in reverse chronological order	37
21.5.	YearCollection.neighbors(): Find previous and next month	37
21.6.	YearCollection.__findPrev(): Find predecessor month	38
21.7.	YearCollection.findNext(): Find successor month	40
22.	class YearRow: Container for one year's records	41
22.1.	YearRow.__init__(): Constructor	42
22.2.	YearRow.__len__(): Number of contained months	42
22.3.	YearRow.__getitem__(): Retrieve one month	43
22.4.	YearRow.firstMonth(): Return the first month	43
22.5.	YearRow.lastMonth(): Return the last month	43
22.6.	YearRow.predecessor(): Find the previous month	44
22.7.	YearRow.successor(): Find the following month	44
22.8.	YearRow.readAllMonths(): Process input files for one year	45
22.9.	YearRow.readOneMonth(): Read one monthly file	46
22.10.	YearRow.writeMonthPages(): Generate monthly pages	47
23.	class MonthCell: One table cell	47
23.1.	MonthCell.__init__(): Constructor	48
23.2.	MonthCell.fileName(): Path to the month's page	48
23.3.	MonthCell.writePage(): Render as XHTML	49
23.4.	MonthCell.__pageFrame(): Set up a basic page	50
23.5.	MonthCell.__renderPage(): Add the notes content	52
23.6.	MonthCell.__pageTOC(): Generate page table of contents	53
23.7.	MonthCell.__dayTOC(): Month table of contents entry for one day	54
23.8.	MonthCell.__notablesBlock(): Display any notable records	56
23.9.	MonthCell.__span(): Add a span inline	57
23.10.	MonthCell.__dayBlock(): Render one daily note set	57
23.11.	MonthCell.__dayTitle(): Render the title for one daily block	58
23.12.	MonthCell.__daySummary(): Render the daily summary block	58
23.13.	MonthCell.__locDef(): Display a locality definition	59
23.14.	MonthCell.__dayAnnotation(): Render day-annotation content	61
23.15.	MonthCell.__annoBlock(): Annotation block with a label	61
23.16.	MonthCell.__narrative(): Render a Narrative instance	62
23.17.	MonthCell.__paragraph(): Render one paragraph	63
23.18.	MonthCell.__paraContent(): Content of one paragraph	63
23.19.	MonthCell.__birdForm(): Render one BirdForm	64
23.20.	MonthCell.__singleSighting(): Single-sighting case	66
23.21.	MonthCell.__multiSighting(): Multiple-sighting case	67
23.22.	MonthCell.__locGroup(): Render a locality group	67
23.23.	MonthCell.__sightNotes(): Render a sighting notes group	68
23.24.	MonthCell.__photo(): Generate a photo reference or link	69
23.25.	MonthCell.__floc(): Generate one of multiple sightings	70
23.26.	MonthCell.__ageSexGroup(): Render age-sex-group content	71
23.27.	MonthCell.monthName(): Translate a month key to a month name (static method)	73
24.	Epilogue	73
25.	Defects discovered	73
25.1.	Syntax errors	74
25.2.	Logic errors	74
25.3.	Run-time type matching errors	76

# 1. Introduction

---

Once birdwatching field notes are encoded in the form described in *A system for encoding bird field notes*<sup>3</sup>, the next link in the tool chain is described herein: rendering these notes into a Web structure.

The *noteweb* script expects to be run from a directory containing all the notes files in this form:

- Each year's worth of notes resides in subdirectory *yyyy/*, where *yyyy* is the four-digit year.
- Each month's worth of notes resides in a single file at path *yyyy/yyyy-mm.xml*, where *mm* is the two-digit, zero-padded month number in the range 01 - 12.

Assuming all the input files are valid, the *noteweb* script builds these files, also in and under the current directory:

- A set of HTML files, one for each input *.xml* file, but named *yyyy/yyyy-mm.html*.
- *field.html*, a thumb-index page with links to the monthly pages. This organized as a table, with the years in rows and the months in columns.

## 2. Operation

---

The operation of the *noteweb* script is straightforward. Once the directories and files are set up as described above, use this command:

```
noteweb [-f|--force]
```

The program will print error messages if any problems are found in the input files. If all are valid, it will build the HTML files.

The *-f* or *--force* option forces all HTML files to be rewritten. By default, the program will rewrite an HTML file only if it has an older modification time than the corresponding XML input file.

### 2.1. Further work

The `birdnotes.py` module now includes a new class `BirdNoteTree` that represents a tree of bird notes as *noteweb* expects it to be structured. Rework *noteweb* to use this module.

## 3. Overview of the internals

---

This document is an example of two software technologies: the Cleanroom development methodology<sup>4</sup> and lightweight literate programming<sup>5</sup>.

The *noteweb* script is written in Python<sup>6</sup> programming language. It is built on these layers:

- The module that handles reading XML bird notes files is described in *A system for encoding bird field notes*<sup>7</sup>. The reader is expected to be familiar with the Relax NG schema described here, as well as the Python interface.

---

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

<sup>4</sup> <http://www.nmt.edu/~shipman/soft/clean/>

<sup>5</sup> <http://www.nmt.edu/~shipman/soft/litprog/>

<sup>6</sup> <http://www.python.org/>

<sup>7</sup> <http://www.nmt.edu/~shipman/aba/doc/>

- Matters of the biological classification of birds are handled by the `txny.py` module described in *A system for representing bird taxonomy*<sup>8</sup>.
- Generation of Web pages in XHTML 1.1 is covered in *Python XML processing with lxml*<sup>9</sup>.
- General page layout and navigation for Web pages follows New Mexico Tech Computer Center practice. Generation of pages in this style is supported by the module described in *tccpage2.py: Dynamic generation of TCC-style web pages with lxml*<sup>10</sup>.

## 4. The generated XHTML

The term “page body” refers to the variable part of a web page. The top level of the page, down to the `body` element, is generated by the `tccpage.2` module discussed in Section 3, “Overview of the internals” (p. 4).

Here is the set of navigational links we will pass to the `TCCPage()` constructor:

- *Next*: For the index page, this is a dead link. For each monthly page except the last, it will point at the next page in chronological sequence.
- *Previous*: For the index page, this is a dead link. For each monthly page but the first, it points at the previous page in sequence.
- *Contents*: For the index page, a dead link. For each monthly page, points back to the index page.
- *Home*: Points to Shipman's homepage.

### 4.1. XHTML for the index page

Here is the body content for the index page.

```
<h1>John W. Shipman's field notes</h1>
<p>
  Notes are grouped according to the reporting seasons for
  <cite>Audubon Field Notes</cite>.
</p>
<p>
  See
  <a href='conventions.html'>How to read Shipman's field notes</a>
  for notational conventions and the author's contact information.
</p>
...index table...
```

The referenced `conventions.html` page is hand-built and outside the scope of this script.

Here is the wrapper for the index table. Its `class='seasons'` attribute links it to the stylesheet rule described in Section 5.3, “`table.seasons`” (p. 13).

```
<table border='5' cellpadding='5' class='seasons' width='100%'>
  <colgroup align='right' />
  <colgroup span='12' align='left' width='*' />
  <thead>
    <tr class='year-row'>
      <th>Year</th>
```

<sup>8</sup> <http://www.nmt.edu/~shipman/xnomo/>

<sup>9</sup> <http://www.nmt.edu/tcc/help/pubs/pylxml/>

<sup>10</sup> <http://www.nmt.edu/tcc/projects/tccpage2>

```

    <th class='winter' colspan='2'>Winter</th>
    <th class='spring' colspan='3'>Spring</th>
    <th class='summer' colspan='2'>Summer</th>
    <th class='fall' colspan='4'>Fall</th>
    <th class='winter'>Winter</th>
  </tr>
</thead>
<tbody>
  ...rows for years here...
</tbody>
</table>

```

The columns of the table are color-coded by season, and the winter season occurs both at the beginning and ending of the year. The months are assigned to seasons by the reporting requirements of *Audubon Field Notes*: winter is December–February; spring is March–May; summer is June–July; and fall is August–November. The color-coding is handled by CSS rules; see Section 5.4, “Seasonal column markup rules: `th.winter`, etc.” (p. 13).

Here is the XHTML for a table row for the year 1990. This illustrates the treatment of months that have no data. The `class='year-group'` attribute appears periodically on the `tr` element to add a heavier ruled line under the row (other `tr` elements have `class='year-row'`). This is a very important user-friendliness feature: without a bold line under a row every so often, and the color-coded columns, the reader's eye can get lost in the grid. The frequency of the `year-group` attribute is given by Section 10.11, “`YEAR_GROUP_FREQUENCY`” (p. 21). For the style markup of table rows, see Section 5.5, “CSS rules for rows of the index table” (p. 14).

```

<tr class='year-group'>
  <th class='row-label'>1990</th>
  <td class='winter'><a href='1990/1990-01.html'>January</a></td>
  <td class='winter'>&nbsp;</td>
  <td class='spring'><a href='1990/1990-03.html'>March</a></td>
  <td class='spring'>&nbsp;</td>
  <td class='spring'><a href='1990/1990-05.html'>May</a></td>
  <td class='summer'>&nbsp;</td>
  <td class='summer'>&nbsp;</td>
  <td class='fall'>&nbsp;</td>
  <td class='fall'>&nbsp;</td>
  <td class='fall'>&nbsp;</td>
  <td class='fall'><a href='1990/1990-11.html'>November</a></td>
  <td class='winter'>&nbsp;</td>
</tr>

```

The year number is in a `th` element styled by a CSS rule described in Section 5.6, “`th.row-label`: The row label” (p. 14). The table cells are in the usual `td` element, marked up with one of the seasonal style rules discussed in Section 5.4, “Seasonal column markup rules: `th.winter`, etc.” (p. 13). Each cell contains a link to the month file if there is one, or a non-breaking space `&nbsp;` placeholder if there is no corresponding month page.

## 4.2. XHTML for the month page

The page's title is “Shipman's field notes, *monthname yyyy*”.

The overall page structure has these parts:

- An h1 heading with a repeat of the page title, followed by a brief introductory paragraph.

```
<h1>Shipman's field notes, September 2007</h1>
<p>
  <a href='http://www.nmt.edu/~shipman/aba/conventions.html'
  >How to read Shipman's field notes</a>
</p>
<ul>
  ...page TOC...
</ul>
<hr/>
...first day-notes...
<hr/>
...second day-notes...
etc.
```

- The *page TOC* (table of contents) has links to each daily section, and also lists the notable records for each day.
- Each *day-notes* element from the source XML file is preceded by a horizontal rule (*hr*). The anchor name for each day's notes is the date in the form "*Dyyyy-mm-dd*".

The schema allows more than one *day-notes* element for the same day. In this case, the anchor for the second and succeeding elements will have the same form except that letters will be appended, e.g., "*D2005-05-16a*", "*D2005-05-16b*", and so forth.

The page table of contents consists of a sequence of *li* (list item) elements, one for each *day-notes* element in the input file.

Each entry in the page TOC starts with a simple link to the anchor for that *day-notes* element. If there were any notable birds inside that element, the entry is followed by a *div* containing the names of the notable birds. Here is an example:

```
<li>
  <a href='#D2007-09-30'>NM: 2007-09-30: Bosque del Apache NWR</a>
  <div class='loc-child'>
    <span class='notable'>Notable:</span>
    Osprey, Solitary Sandpiper, Sabine's Gull, Common Tern
  </div>
</li>
```

The notable block is formatted by a CSS rule discussed in Section 5.7, "*div.loc-child: Indented child block*" (p. 14). For styling of the label "Notable:", see Section 5.2, "Inline markup rules" (p. 12).

### 4.3. XHTML for one day-notes

Here is an outline of the XHTML generated for one *day-notes* element.

- A horizontal rule, "*<hr/>*".
- An h2 heading whose *id* attribute is the anchor used in the page TOC, and whose text has this form:

```
<h2 id='yyyy-mm-dd'>rr: yyyy-dd-mm: day-loc</h2>
```

where *yyyy-mm-dd* is the date, *rr* is the region (state) code, and *day-loc* is the locality string that covers the whole *day-notes* element.

- A `div` that wraps the daily summary block. This uses the CSS rule discussed in Section 5.8, “`div.day-summary: Daily summary block`” (p. 15). See Section 4.4, “XHTML for the day-summary block” (p. 8).
- The `form` elements from the input, in phylogenetic order. See Section 4.8, “XHTML for the form element” (p. 9).

## 4.4. XHTML for the day - summary block

The daily summary block is wrapped in a “`<div class='day - summary '>`” element. It contains:

- A repeat of the list of notable species that appears in the page TOC, if there are any.
- See the Section 4.4.1, “Historical note” (p. 8).
- Blocks showing each locality code and its definition; see Section 4.5, “XHTML for locality definitions” (p. 8).

A table isn't really the right presentation for the locality code definitions, because there are two kinds of optional content (narrative and GPS waypoint blocks) that don't occur often enough to rate their own table columns. Putting them in a spanned row would disrupt the table structure.

- Blocks for any of `day - annotation` elements that may be present; see Section 4.6, “XHTML for day - annotation elements” (p. 9).

### 4.4.1. Historical note

In the output of the XSLT script that was the predecessor of *noteweb* the default locality code was displayed before the list of localities. For a discussion of why this no longer appears, see the remarks in the Section 23.12.1, “Historical note” (p. 59) accompanying Section 23.12, “`MonthCell.__daySummary()`: Render the daily summary block” (p. 58). In any case, the output was:

```
<div class='loc-child'>
  <span class='loc-label'>
    Default location: @code
  </span>
</div>
```

where *code* is the default locality code for the day.

## 4.5. XHTML for locality definitions

For each defined locality code, this XHTML is generated:

```
<div class='loc-def'>
  @code: name
  <div class='loc-narrative'>
    text
  </div>
  <div class='loc-narrative'>
    GPS: waypoint narrative
  </div>
  ...
</div>
```

where *code* is the locality code and *name* is the name attribute of the locality. For the styling of the outer `div`, see Section 5.9, “`div.loc-def: Location definition`” (p. 15).

The first inner `div` is generated only when there is a long description for the locality. If there are GPS waypoints defined for this locality, they follow inside the outer `div`. For the styling of these blocks, see Section 5.10, “`div.loc-narrative`: Locality narrative” (p. 15).

## 4.6. XHTML for day-annotation elements

Still inside the daily summary block, here is the XHTML generated for elements in the `day-annotation` pattern: `route`, `weather`, `missed`, `film`, and any undifferentiated `para` elements or loose text children of the `day-summary` element.

```
<div class='loc-child'>
  <span class='loc-label'>type:</span>
  text
</div>
```

The `span` portion is omitted for undifferentiated text; otherwise it shows the content type such as “Film”.

The `text` may be a single paragraph, or it may be broken into multiple paragraphs. See Section 4.7, “XHTML rendering of narrative elements” (p. 9).

If there are multiple paragraphs of narrative, each is in a separate `div` below the run-in label. However, if there is only one paragraph, that is a special case, and the content of that one paragraph is placed directly in the parent `div`.

## 4.7. XHTML rendering of narrative elements

An instance of the `Narrative` class from the `birdnotes.py` module is basically a container for one or more `Paragraph` instances.

A `Paragraph` instance, in turn, is a container for a sequence of text strings, each of which may or may not be marked up with either a `genus` or `cite` tag. The XHTML rendering of a `Paragraph` is:

```
<div class='para'>
  text
</div>
```

where `text` is a mixture of untagged text and text wrapped in `span` elements. For the styling of this block, see Section 5.11, “`div.para`: Ordinary paragraphs” (p. 15).

Input text marked up with a `genus` tag is wrapped like this:

```
<span class='genus'>...</span>
```

Text marked with `cite` will look like this:

```
<span class='cite'>...</span>
```

## 4.8. XHTML for the form element

Each `form` element from the input file is rendered like this:

1. A `div` element with `class='form'` wraps all generated content. For styling, see Section 5.12, “`div.form`: General form-related data” (p. 16).

Because the bird names of notable records are set in a red-bordered box, they need a little bit more vertical margin, so there is a variant rule for `class= 'notable-form'`.

2. Display of the name or names of this form of bird. For the display of the bird name(s), see Section 4.9, “XHTML for bird form names” (p. 10).
3. If there is locality information attached to the `form` element and `loc-group` or `sighting-note` content attached, it is rendered next. See Section 4.10, “XHTML rendering of form data” (p. 10).
4. If the `form` element has `floc` children, they are rendered next. See Section 4.13, “XHTML rendering of `floc` elements” (p. 12).

## 4.9. XHTML for bird form names

Rendering of the bird name is driven by a `BirdForm` instance from `birdnotes.py`. This instance's `.birdId` attribute is an instance of class `BirdId` from the taxonomy module `txny.py`; it can represent single taxa, hybrids, or species pairs such as “Dusky/Hammond's Flycatcher.”

The entire name is wrapped in:

```
<span class='bird-name'>...</span>
```

Inside this `span` are placed:

1. The English name of the first or only form.
2. If this is a compound form, the string ' × ' for hybrids or ' / ' for pair forms, followed by the English name of the second form.
3. If the record is questionable, “?” is appended; if uncountable, “[uncountable]” is appended.

## 4.10. XHTML rendering of form data

Elements at two levels—`form` and `floc`—share the need to display two kinds of data that can be attached at either level: `loc-group` content and `sighting-notes` content. A third type of content, `age-sex-group`, can appear at either level in the XML input, but in the representation provided by the `birdnotes.py` module, it is always attached to the `Sighting` child of the `BirdForm` element.

The overall rendering of an input `form` element depends on whether it is the single-sighting or multi-sighting case:

- In the single-sighting case, let us refer to that single sighting as *S*. There is only one `div` at the top level, and it looks schematically like this.

```
<div class='form'>
  <span class='bird-name'>Greater Roadrunner</span>
  S.age-sex-group
  S.loc-group
  S.sighting-notes
</div>
```

where the `loc-group` and `sighting-notes` are rendered in separate `div` elements, but the `age-sex-group` follows the bird name on the same line.

In this case, the `loc-group` may be inherited all or partially from the `day-notes` parent.

- In the multiple-sighting case, here is a schematic of the rendering, where *F* is the `form` input element and *S* is a child sighting element.

```

<div class='form'>
  <span class='bird-name'>Greater Roadrunner</span>
  F.loc-group
  F.sighting-notes
  <div class='floc'>
    S.age-sex-group
    S.loc-group
    S.sighting-notes
  </div>
  <div class='floc'>
    ...
  </div>
  ...
</div>

```

In this case, the *F.loc-group* part appears only if there is an explicit `.locGroup` attribute on *F*. There is no point in using the inherited value, since the inherited parts will be displayed in the child sightings, combined with any explicit locality data in the children.

Here are some notes on the rendering of these various elements.

- If there is a count in the `age-sex-group`, it is copied to the XHTML, followed by a space.
- If there is an age code, it is copied directly, except that code “p” is converted to a Greek lowercase phi,  $\phi$ , as entity reference “&#x03c6;”.
- If there is a sex code, it is copied directly.
- If the countability code is “?”, it is copied directly; countability code “-” is rendered as “[uncountable]”.
- If the record is *fide* some other observer, that is rendered as “[`<span class='genus'>fide</span> N]`”, where *N* is the observer's name. Although the word *fide* is not technically a genus name, it falls under the same rule: italicize Latin words.
- The `locality-group`, materialized as a `LocGroup` instance, generates both inline and block content.
  - There is always a locality code, explicit or inherited, and we translate that to the corresponding full name, preceded by an “@” sign.
  - If there is a GPS waypoint, we generate the inline text “GPS=...”.
  - If there is `loc-detail` content, it is rendered in a separate `div` with `class='loc-child'`. For the associated CSS rule, see Section 5.7, “`div.loc-child`: Indented child block” (p. 14).
- For the rendering of the various `sighting-notes` elements, see Section 4.11, “XHTML rendering of sighting notes” (p. 11).

## 4.11. XHTML rendering of sighting notes

The elements of the `sighting-notes` content are rendered in this order:

- If there is a description, it is packaged in a `div class='loc-child'`. The content is prefaced with the text “Description:”, with that text wrapped in a `span class='loc-label'`.
- If there is a behavior block, it is packaged in a `div` like the previous case, except that the labeling text is “Behavior:”.
- If there is a vocalization block, it is packaged like the previous cases, with the label “Vocalization:”.
- If there are breeding notes, they are packaged the same way, with the label “Breeding:”.

- There may be any number of photo links; these are rendered next. See Section 4.12, “XHTML rendering of photo links” (p. 12).
- General narrative follows. For its rendering, see Section 4.7, “XHTML rendering of narrative elements” (p. 9).

## 4.12. XHTML rendering of photo links

Each photo link is rendered one of two ways, depending on whether there is a `url` attribute.

- If there is a `url`:

```
<a href='url'>
  <img src='/~shipman/thumb/catNo.jpg' alt='catNo' />
</a>
```

where *catNo* is the photo's catalog number.

- If there is no `url`, the catalog number is copied to the page, unadorned.

## 4.13. XHTML rendering of floc elements

Each input `floc` element is wrapped in an XHTML `div` element with `class='floc'`. For the associated style rule, see Section 5.13, “`div.floc`: Multiple sightings” (p. 16).

The rendering of the sighting proceeds the same as described in Section 4.10, “XHTML rendering of form data” (p. 10).

# 5. The style sheet

---

The style sheet displayed here is the actual CSS (Cascading Style Sheets, Level 2) file, in literate-programming form.

birdnotes.css

```
/* Style sheet for bird notes. Do not edit this file directly;
 * it is generated automatically from this documentation:
 * http://www.nmt.edu/~shipman/aba/doc/noteweb/
 */
```

## 5.1. General style

The author prefers a pale yellow background and not a stark white.

birdnotes.css

```
body
{ background-color: #ffffee; /* Very pale yellow */
}
```

## 5.2. Inline markup rules

Here are some rules for styling inline content. The `span.loc-label` rule is used to italicize inline labels that set off paragraphs.

```
span.loc-label
{
  font-style: italic;
}
```

The `span.bird-name` rule styles species names. The text is dark green, and boldfaced.

```
span.bird-name
{
  color: #007722;           /* Dark green */
  font-weight: bold;
}
```

The label “Notable:” needs to stand out, so we set it in maroon type, surrounded by a thin red box.

```
span.notable
{
  font-weight: bold;
  color: maroon;
  border-style: solid;
  border-width: 1px;
  border-color: red;
  padding: 1px;
}
```

These two rules are used to mark up genus names and literature citations, both of which are traditionally italicized.

```
span.genus
{
  font-style: italic;
}
span.cite
{
  font-style: italic;
}
```

### 5.3. table.seasons

In order to use borders inside table rows and cells, we must select the “separate borders” table model for the index page.

```
table.seasons
{
  border-collapse: collapse;
}
```

### 5.4. Seasonal column markup rules: th.winter, etc.

Columns in the index table are color-coded according to the four reporting seasons for *Audubon Field Notes*. We provide rules for both heading (`th`) and normal (`td`) table cells. The class names are the season names. All cells have borders on the left and right, so that the entire table is ruled.

```
th.winter, td.winter
{
  background-color: #befaff; /* ice blue */
  border-left: 1px solid black;
}
```

```

    border-right: 1px solid black;
}
th.spring, td.spring
{  background-color: #9affa5; /* spring green */
  border-left: 1px solid black;
  border-right: 1px solid black;
}
th.summer, td.summer
{  background-color: #fff870; /* pale yellow */
  border-left: 1px solid black;
  border-right: 1px solid black;
}
th.fall, td.fall
{  background-color: #eeaa3a; /* dark orange */
  border-left: 1px solid black;
  border-right: 1px solid black;
}

```

## 5.5. CSS rules for rows of the index table

These rules style the rows in the index table. We want rules around the bottom, left, and right sides of the row.

birdnotes.css

```

tr.year-row
{  border-bottom: 1px solid black;
  border-left: 1px solid black;
  border-right: 1px solid black;
}

```

To help the reader's eye track across the table, we make every fifth row's bottom border thicker. The `year-group` class is added for rows whose years are evenly divisible by 5.

birdnotes.css

```

tr.year-group
{  border-bottom: 3px solid black;
}

```

## 5.6. `th.row-label`: The row label

This rule is used to boldface the year that labels each row of the index table.

birdnotes.css

```

th.row-label
{  font-weight: bold;
}

```

## 5.7. `div.loc-child`: Indented child block

This style element is used for children of the daily summary, and in other places where we want to set an indented block off with a little more vertical space, and use a hanging indent so continuation lines are indented further.

```
div.loc-child
{
  margin-top: 0.25em;
  margin-bottom: 0.25em;
  margin-left: 3em;
  text-indent: -1em;
}
```

## 5.8. div.day-summary: Daily summary block

The entire daily summary block is set inside a thin solid border, on a light gray background.

```
div.day-summary
{
  margin-left: 1em;           /* Indent on left */
  margin-bottom: 1em;        /* Add space below */
  border-style: solid;       /* Solid border */
  border-width: 1px;         /* One pixel wide */
  padding: 10px;             /* Leave room inside the box */
  background-color: #eee;    /* Light gray background */
}
```

## 5.9. div.loc-def: Location definition

This rule styles the block containing the definition of a locality code. The block is indented and boldfaced.

```
div.loc-def
{
  margin-left: 2em;
  font-weight: bold;
}
```

## 5.10. div.loc-narrative: Locality narrative

The narrative that describes a locality should be indented further than its parent, but with a “hanging indent” so that continuation lines, if any, will be indented even further.

```
div.loc-narrative
{
  margin-left: 3em;
  margin-top: 3px;
  text-indent: -1em;
  font-weight: normal;
}
```

## 5.11. div.para: Ordinary paragraphs

This rule styles ordinary paragraphs in any narrative content.

```
div.para
{
  margin-left: 1em;
}
```

```
margin-top: 0.5em;
}
```

## 5.12. `div.form`: General form-related data

This rule styles the block for each form of bird. The block is indented, with a bit of space above, and hanging indentation so the first line is not indented.

birdnotes.css

```
div.form
{ margin-top: 0.1em;
  margin-left: 1em;
  text-indent: -1em;
}
```

Because notable records are displayed in a red-bordered box, they need a bit more margin, or adjacent boxes will overlap. Hence this variant of the above rule:

birdnotes.css

```
div.notable-form
{ margin-top: 0.5em;
  margin-bottom: 0.5em;
  margin-left: 1em;
  text-indent: -1em;
}
```

## 5.13. `div.floc`: Multiple sightings

This rule applies to the rendering of sighting data when there are multiple sightings for a given kind of bird. We indent these sightings an additional 2 ems.

birdnotes.css

```
div.floc
{ margin-left: 2em;
}
```

## 6. Design notes

---

Before we examine the actual *noteweb* script, a few comments on data structures and algorithms are in order.

Because of the need for navigational links between pages, we can't just go out and find monthly XML files and immediately convert them to HTML. Each monthly page must have *Next* and *Previous* navigational links. So, when we build a monthly page, we need to know which month (if any) was the previous one in sequence, and which is the next in sequence. There is no guarantee that every month has a valid input file. There might even be years with no valid input files.

Therefore, the first thing we have to do is read all the XML files, rendering each one into a `bird-notes.BirdNoteSet` instance. Then we can work through these instances, converting each to an HTML page in the same subdirectory where we found the XML input file. (Note that keeping all these `BirdNoteSet` instances around may eat up a lot of memory. If that is ever a problem, we'll just have to make two passes: once to see which files are valid, and another pass to render them, so that we don't have to keep the entire data set in memory at once.)

We must also generate the index page, with a table of links to all the months. Each row in this table contains all the months of that year. There is, however, no guarantee that years are contiguous. We just look to see what year directories are present, and that determines the set of table rows.

The above conditions suggest a data structure made from instances of three classes:

1. One `YearCollection` instance contains everything we need to build the index page.

Because this instance contains all the input data, it can figure out which months are the *Previous* and *Next* navigational links for a given month.

2. The `YearCollection` instance is a container for `YearRow` instances, one for each year for which there is an input data directory.

Each `YearRow` instance has all the information needed to build one row of the index table.

3. Each `YearRow` instance is a container for up to twelve `MonthCell` instances.

Each `MonthCell` instance has all the information about one month for which there is an input XML file, and has everything needed to build the monthly HTML page.

## 6.1. Discarded approaches

The first cut at an overall data structure was a list named `yearList` containing `YearRow` instances. Each `YearRow` would be a container for `BirdNoteSet` instances, one per valid month.

However, there are certain things we need to know about the months, such as the month number (e.g., '04'). So the `MonthCell` class was invented, with an instance holding one `BirdNoteSet` and ancillary information such as the month number and the file name of the month page. Each `YearRow` would then be a container for `MonthCell` instances, one per valid month.

The next problem was connecting up the navigation links between month pages. Clearly, when rendering a month page, we must know the URL of the *Previous* month page (if any) and the *Next* month page (if any). However, how does the `MonthCell` instance know these URLs? Three approaches were considered:

- Let the `MonthCell` class have attributes that hold the previous/next links; initialize them to `None`.

Then, after all the `MonthCell` instances are created, make a serial pass through them and link them up into a bidirectional linked list.

Finally, render each `MonthCell` into HTML in any old order, using the stored previous/next attributes to set up its navigation.

The aesthetic objection to this approach was that the `MonthCell` objects depend on an external mechanism to set up their linking. Logically, the information required to find a month's neighbors should reside at a higher organizational level.

- Define a global function called something like `neighbors()` that finds the previous/next neighbors, given a year and month. This function would walk through the `yearList`, starting at the given month, and working backwards and forwards to find the nearest neighbors.

The objection to this approach is that the `yearList` must then be global, or be passed around through many levels.

- At this point, the author felt that a third class was called for: `YearCollection`, which manages the overall sequence of years *and* months.
  - It can traverse the years in reverse chronological order to build the index table with the most recent years first.
  - It is the obvious place for logic that can find the neighbors of any month.

## 7. Overall program flow

---

Here is the overall flow of the program:

1. Survey the current working directory for subdirectories with year names in the range 1000-2999, and build a list of `YearRow` instances.

For each year, look in that subdirectory for monthly files with names like `yyyy-mm.xml`. If the file is unreadable or invalid, spew appropriate error messages. Otherwise, convert the file into a `birdnotes.BirdNoteSet` instance and store it inside the year's `YearRow` instance.

2. Sort the list of `YearRow` instances into reverse chronological order.
3. Work through the list of `YearRow` instances. For each contained month, render the `BirdNoteSet` instance into XHTML.
4. Once all the monthly pages are built, write the completed index page.

## 8. Prologue

---

Here we start the actual code of the `noteweb` script.

```
noteweb
#!/usr/bin/env python
#=====
# noteweb: Render XML bird field notes into HTML
#   For documentation, see:
#     http://www.nmt.edu/~shipman/aba/doc/noteweb/
#-----
PROGRAM_NAME = "noteweb"
EXTERNAL_VERSION = "1.1"
```

## 9. Imported modules

---

For documentation on these modules, see Section 3, “Overview of the internals” (p. 4).

The standard Python `sys` module gives us command line arguments and I/O streams. The `os` module allows us to crawl around the directory structure, build and dismantle path names, and so forth. The `stat` module allows access to file modification timestamps. Module `re` is the standard Python regular expression module. The `optparse` module parses command line arguments.

```
noteweb
#=====
# Imports
#-----
import sys, os, stat, re, optparse
```

The `Singleton` class comes from *Scanning objects for Python: Scan and Log*<sup>11</sup>.

```
noteweb
from singleton import Singleton
```

The `birdnotes.py` module takes care of reading the XML field notes input files.

---

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

```
import birdnotes
```

Because we'll need to sort records into phylogenetic order, we import the bird taxonomy module.

```
import txny as taxonomy
```

The `lxml.etree` module handles generation of XHTML. Because this is used heavily, we abbreviate it as `et`.

```
import lxml.etree as et
```

General page structure and navigation is set up by `tccpage2.py`.

```
import tccpage2
```

## 10. Manifest constants

By convention, values constant in this program carry names all in capital letters, with underbar (`_`) as the word separator.

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

### 10.1. MONTH\_NAME\_MAP: Translate month numbers to month names

Internally we carry month numbers such as `'03'`. This dictionary translates month numbers to month names.

```
MONTH_NAME_MAP = {
    '01': 'Jan', '02': 'Feb', '03': 'Mar',
    '04': 'Apr', '05': 'May', '06': 'Jun',
    '07': 'Jul', '08': 'Aug', '09': 'Sep',
    '10': 'Oct', '11': 'Nov', '12': 'Dec' }
```

### 10.2. MONTH\_SEASON\_MAP: Define the season for each month

The *Audubon Field Notes* reporting seasons are tied to months of the year. This dictionary translates month numbers into season names, which in turn connect the rendering of the index table to the seasonal column color codes defined in Section 5.4, “Seasonal column markup rules: `th.winter`, etc.” (p. 13).

```
MONTH_SEASON_MAP = {
    '01': 'winter', '02': 'winter', '03': 'spring',
    '04': 'spring', '05': 'spring', '06': 'summer',
    '07': 'summer', '08': 'fall', '09': 'fall',
    '10': 'fall', '11': 'fall', '12': 'winter' }
```

### 10.3. YEAR\_PAT: Pattern for year numbers

This pattern matches year numbers in the range 1000-2999. The '\$' character is the end-of-line anchor, and matches only at the end of a string.

noteweb

```
YEAR_PAT = re.compile (
    r'[12]'      # First digit must be 1 or 2
    r'\d{3}'    # Followed by three more digits
    r'$' )     # Insure that the entire string was matched
```

### 10.4. YYYY\_MM\_XML\_PAT: Month file name pattern

This pattern matches the name of a monthly file such as '2005-09.xml'. Note that it will match invalid month names such as '1999-00.xml' and '2008-19.xml', but that is hardly a world-shattering defect.

noteweb

```
YYYY_MM_XML_PAT = re.compile (
    r'[12]\d{3}' # Matches 1000-2999
    r'\-'       # Matches '-'
    r'[01]\d'   # Matches 00-19
    r'\.'       # Matches '.'
    r'xml'      # Matches 'xml'
    r'$' )     # End-of line anchor: insure a complete match
```

### 10.5. HTML\_EXT: File extension for XHTML pages

This string is appended to "yyyy-mm" month names to form the name of each generated monthly report page.

noteweb

```
HTML_EXT = '.html'
```

### 10.6. INDEX\_PAGE\_NAME: Name of the index page

This string, with HTML\_EXT appended, is the name of the page to be generated containing the index table.

noteweb

```
INDEX_PAGE_NAME = 'field'
```

### 10.7. INDEX\_PAGE\_TITLE

Title of the index page.

noteweb

```
INDEX_PAGE_TITLE = "John W. Shipman's Field Notes"
```

## 10.8. HOME\_PAGE\_URL: Home page URL

noteweb

```
HOME_PAGE_URL = 'http://www.nmt.edu/~shipman/'
```

## 10.9. CONVENTIONS\_URL

This is the URL of the page explaining the conventions used in the field notes display.

noteweb

```
CONVENTIONS_URL = '/~shipman/aba/'
```

## 10.10. SEASONS\_CLASS: Class attribute for the table of seasons

This class name is used to link the index table to the rule given in Section 5.3, “table.seasons” (p. 13).

noteweb

```
SEASONS_CLASS = 'seasons'
```

## 10.11. YEAR\_GROUP\_FREQUENCY

This constant determines how often a year's row in the table has a thicker lower border. For example, if the value is 5, this groups years visually into groups of five, so the reader's eye can more easily track across the row.

noteweb

```
YEAR_GROUP_FREQUENCY = 5
```

## 10.12. YEAR\_GROUP\_CLASS: CSS class for years ending a year group

See Section 10.11, “YEAR\_GROUP\_FREQUENCY” (p. 21) for an explanation of year groups. This value is the CSS stylesheet class to be applied to year rows that are a multiple of YEAR\_GROUP\_FREQUENCY. This class name is tied to the CSS rule in Section 5.5, “CSS rules for rows of the index table” (p. 14).

noteweb

```
YEAR_GROUP_CLASS = 'year-group'
```

## 10.13. YEAR\_ROW\_CLASS

CSS link for rows in the index table for normal years.

noteweb

```
YEAR_ROW_CLASS = 'year-row'
```

## 10.14. ROW\_LABEL\_CLASS

This constant defines the CSS class name for the row labels in the index table. See Section 5.5, “CSS rules for rows of the index table” (p. 14).

noteweb

```
ROW_LABEL_CLASS = 'row-label'
```

## 10.15. NBSP: Non-breaking space character

The entity `&nbsp;` has the Unicode position `&#x00a0;`.

noteweb

```
NBSP = u'\u00a0'
```

## 10.16. PHI: Greek letter

Greek lowercase phi, used to denote an age class of “female or immature.” Symbolically this entity is “`&phiv`”.

noteweb

```
PHI = u'\u03d5'
```

## 10.17. CSS\_URL: Our stylesheet

This is the URL of the stylesheet for the generated pages.

noteweb

```
CSS_URL = 'http://www.nmt.edu/~shipman/aba/doc/noteweb/birdnotes.css'
```

## 10.18. ZDP\_LOGO

Escape from the weight of your corporate logo.

—Frank Zappa

noteweb

```
ZDP_LOGO = 'http://www.nmt.edu/~shipman/zdplogo.png'
```

## 10.19. ZDP\_URL

The homepage for navigational links.

noteweb

```
ZDP_URL = 'http://www.nmt.edu/~shipman/z'
```

## 10.20. LOC\_CHILD\_CLASS

Used to link to the CSS rule in Section 5.7, “`div.loc-child`: Indented child block” (p. 14).

noteweb

```
LOC_CHILD_CLASS = 'loc-child'
```

## 10.21. NOTABLE\_CLASS

Used to link a `span` element to highlight notable records; see Section 5.2, “Inline markup rules” (p. 12).

noteweb

```
NOTABLE_CLASS = 'notable'
```

## 10.22. DAY\_SUMMARY\_CLASS

CSS class for daily summary blocks; see Section 5.8, “div.day-summary: Daily summary block” (p. 15).

noteweb

```
DAY_SUMMARY_CLASS = 'day-summary'
```

## 10.23. LOC\_DEF\_CLASS

CSS class for locality definitions; see Section 5.9, “div.loc-def: Location definition” (p. 15).

noteweb

```
LOC_DEF_CLASS = 'loc-def'
```

## 10.24. LOC\_NARRATIVE\_CLASS

CSS class for narrative about a locality; see Section 5.10, “div.loc-narrative: Locality narrative” (p. 15).

noteweb

```
LOC_NARRATIVE_CLASS = 'loc-narrative'
```

## 10.25. LOC\_LABEL\_CLASS

CSS class for run-in paragraph labels; see Section 5.2, “Inline markup rules” (p. 12).

noteweb

```
LOC_LABEL_CLASS = 'loc-label'
```

## 10.26. PARA\_CLASS

CSS class for an ordinary text paragraph; see Section 5.11, “div.para: Ordinary paragraphs” (p. 15).

noteweb

```
PARA_CLASS = 'para'
```

## 10.27. FORM\_CLASS

CSS class for the div that wraps all data for a single bird form. See Section 5.12, “div.form: General form-related data” (p. 16).

noteweb

```
FORM_CLASS = 'form'
```

## 10.28. NOTABLE\_FORM\_CLASS

A variant for notable records that gives more space.

noteweb

```
NOTABLE_FORM_CLASS = 'notable-form'
```

## 10.29. BIRD\_NAME\_CLASS

CSS class for the `span` that wraps the name of a form of bird.

noteweb

```
BIRD_NAME_CLASS = 'bird-name'
```

## 10.30. FLOC\_CLASS

CSS class for the `div` element that wraps each sighting in the multi-sighting case.

noteweb

```
FLOC_CLASS = 'floc'
```

## 10.31. GENUS\_CLASS

CSS class for Latin names; see Section 5.2, "Inline markup rules" (p. 12).

noteweb

```
GENUS_CLASS = 'genus'
```

# 11. main(): The main program

---

For design notes, see Section 6, "Design notes" (p. 16); for the overall flow, see Section 7, "Overall program flow" (p. 18).

Here is the main, and the intended function for the whole program:

noteweb

```
# - - -   m a i n

def main():
    '''Main program.
    [ file 'aou.xml' is a readable, valid taxonomy file ->
      if all monthly files under the current directory are
      valid against birdnotes.rnc ->
        monthly web pages := rendering of those monthly
                           files into XHTML, in the same directory, with
                           the names changed from .xml to .html
        index page := table of links to monthly web pages
    else ->
        sys.stderr += error message(s)
        (monthly web pages, index page) := (anything) ]
    ...
```

A `taxonomy.Txny` instance will be necessary to represent a phylogenetic arrangement of the birds. This constructor assumes the existence of an `aou.xml` file in the current directory; see *A system for representing bird taxonomy*<sup>12</sup>.

noteweb

```
#-- 1 --
# [ if the command line arguments are valid ->
#   Args() := a single Args instance representing those
```

<sup>12</sup> <http://www.nmt.edu/~shipman/xnomo/>

```

#             arguments
#   else ->
#     sys.stderr += usage message
#     stop execution ]
Args()

#-- 2 --
# [ if there is a readable aou.xml in the current directory
#   that is valid against txny.rnc ->
#   txny := a Txny instance representing that file
#   else ->
#     sys.stderr += error message(s)
#     stop execution ]
try:
    txny = taxonomy.Txny()
except IOError, detail:
    print >>sys.stderr, ( "*** Can't read the taxonomy file: %s" %
                          detail )
    raise SystemExit

```

The first step is to create the `YearCollection` instance that will hold all the data. See Section 21, “class `YearCollection`: The top-level data structure” (p. 35).

noteweb

```

#-- 3 --
# [ yearCollection := a new, empty YearCollection instance ]
yearCollection = YearCollection()

```

Next, we find all the subdirectories that look like year directories, and add them to `yearCollection`. See Section 12, “findYears: Locate year directories” (p. 26).

noteweb

```

#-- 4 --
# [ yearCollection := yearCollection with years added
#   corresponding to subdirectories of '.' with year
#   names 1000-2999 ]
#   sys.stderr += error message(s) from invalid monthly XML
#   files in those subdirectories
findYears ( txny, yearCollection )

```

For each year in `yearCollection`, we look for monthly XML files in the corresponding directory. Invalid XML files result in error messages. Valid ones are rendered into HTML, and each month's parent `yearRow` instance is informed of its existence. See Section 21.4, “`YearCollection.genYearsRev()`: Generate years in reverse chronological order” (p. 37) and Section 22.10, “`YearRow.writeMonthPages()`: Generate monthly pages” (p. 47).

noteweb

```

#-- 5 --
# [ monthly web pages := XHTML renderings of BirdNoteSet
#   instances from yearCollection ]
for yearRow in yearCollection.genYearsRev():
    yearRow.writeMonthPages()

```

Now that we know the set of valid months, we can build the index page. See Section 13, “`buildIndex()`: Build the index page” (p. 27).

```
#-- 6 --
# [ index page := table of links to monthly web pages
#       from yearCollection ]
buildIndex ( yearCollection )
```

## 12. findYears: Locate year directories

This function looks for the year subdirectories under the current working directory.

```
# - - -   f i n d Y e a r s

def findYears ( txny, yearCollection ):
    '''Build the list of years.

    [ let
        year-dirs == (subdirectories of '.' with four-digit
            names in the range 1000-2999)
    in:
        (txny is a taxonomy.Txny instance) and
        (yearCollection is a YearCollection instance) ->
        sys.stderr += error messages from invalid monthly
            XML files in year-dirs
        yearCollection := yearCollection with years added
            corresponding to year-dirs ]
    ...
```

We use `os.listdir()` to get a list of all the names in the current working directory, filtering those names with the `YEAR_PAT` regular expression to find the ones that look like year names; see Section 10.3, “`YEAR_PAT`: Pattern for year numbers” (p. 20). Each year directory name is then passed to the `YearRow` constructor, so that `yyyyList` is now a list of `YearRow` instances; see Section 22, “`class YearRow`: Container for one year’s records” (p. 41).

```
#-- 1 --
# [ yyyyList := subdirectories of '.' that match YEAR_PAT ]
yyyyList = [ dirName
              for dirName in os.listdir('.')
              if YEAR_PAT.match(dirName) is not None ]
```

Next we’ll sort the list in chronological order.

```
#-- 2 --
# [ yyyyList := yyyyList sorted in ascending order ]
yyyyList.sort()
```

See Section 21.2, “`YearCollection.addYear()`: Add a year” (p. 36) and, for the logic that searches a year directory for monthly XML files, see Section 22.8, “`YearRow.readAllMonths()`: Process input files for one year” (p. 45).

```
#-- 3 --
# [ yearCollection := yearCollection with valid monthly
```

```

#      data added from XML files in its members' subdirectories
#  sys.stderr += error message(s) about invalid monthly
#      XML files in that set, if any ]
for yyyy in yyyyList:
  #-- 3.1 --
  # [ yearCollection := yearCollection with a new YearRow
  #      added for year=yyyy
  #  yearRow := that new YearRow instance ]
  yearRow = yearCollection.addYear ( txny, yyyy )

  #-- 3.2 --
  # [ yearRow := yearRow with MonthCell instances added
  #      for XML notes files in directory yearRow.yyyy
  #      valid against birdnotes.rnc and txny
  #  sys.stderr += error message(s) for invalid XML
  #      notes files in that directory, if any ]
  yearRow.readAllMonths()

```

## 13. buildIndex ( ): Build the index page

This function builds the index page containing the table of links to the monthly pages. For a summary of the XHTML on this page, see Section 4.1, “XHTML for the index page” (p. 5).

noteweb

```

# - - -   b u i l d I n d e x

def buildIndex ( yearCollection ):
  '''Build the index page.

  [ yearCollection is a YearCollection instance ->
    index page += table of links to monthly web pages
    made from yearCollection ]
  ...

```

First we set up an empty page with appropriate navigational links; see Section 14, “pageFrame: Set up navigation for the index page” (p. 28).

noteweb

```

#-- 1 --
# [ page := a new tccpage2.TCCPage instance with navigation
#      set up for the index page ]
page = pageFrame()

```

By “boilerplate” we mean the fixed content that always starts off the index page. See Section 15, “indexBoilerplate: Fixed content for the index page” (p. 29).

noteweb

```

#-- 2 --
# [ page.content += boilerplate content for the index page ]
indexBoilerplate ( page.content )

```

For the logic that builds the actual index, see Section 16, “indexTable ( ): Build the table of monthly links” (p. 29).

```
#-- 3 --
# [ page.content += the index table made from yearCollection ]
indexTable ( page.content, yearCollection )
```

All that remains is to add a link to the documentation, and write the page where it goes.

```
#-- 4 --
# [ page.content += a link to this document ]
p = et.SubElement ( page.content, 'p' )
p.text = "These pages are generated automatically; see "
docLink = et.SubElement ( p, 'a',
href='http://www.nmt.edu/~shipman/aba/doc/noteweb/' )
docLink.text = "the documentation"
docLink.tail = "."

#-- 5 --
# [ INDEX_PAGE_NAME+HTML_EXT can be created new ->
#   that file := XHTML serialization of page ]
try:
    indexName = INDEX_PAGE_NAME + HTML_EXT
    indexFile = open ( indexName, 'w' )
    page.write ( indexFile )
    indexFile.close()
except IOError, detail:
    print >>sys.stderr, ( "*** Can't write the index page "
        "'%s'." % indexName )
```

## 14. pageFrame: Set up navigation for the index page

This function instantiates a new `tccpage2.TCCPage` instance with navigational links appropriate for the index page. These links should be similar to the ones generated in Section 23.4, “`MonthCell.__page-Frame()`: Set up a basic page” (p. 50).

- *Next*, *Prev*, and *Contents* are dead links, since this is a top-level page.
- *Home* points to Shipman's homepage.

```
# - - -   P a g e   F r a m e

def pageFrame():
    '''Sets up the tccpage2.TCCPage instance for the index page.

    [ return a new tccpage2.TCCPage instance with navigation
      set up for the index page ]
    ...
#-- 1 --
# [ navList := a list of tccpage2.NavLink instances
#   representing the navigational features ]
navList = (
    tccpage2.NavLink ( 'Next', [] ),
    tccpage2.NavLink ( 'Previous', [] ),
```

```

tccpage2.NavLink ( 'Contents', [] ),
tccpage2.NavLink ( 'Home',
  [ ("Shipman's Home Sweet Homepage", HOME_PAGE_URL) ] ) )

#-- 2 --
return tccpage2.TCCPage ( INDEX_PAGE_TITLE, navList,
  logoImage=ZDP_LOGO, logoLink=ZDP_URL, cssUrl=CSS_URL )

```

## 15. indexBoilerplate: Fixed content for the index page

noteweb

```

# - - -   i n d e x B o i l e r P l a t e

def indexBoilerplate ( parent ):
  '''Add the fixed content to the index page.

  [ parent is an et.Element ->
    parent += fixed content for the index page ]
  ...

#-- 1 --
# [ parent += paragraph about seasons ]
p1 = et.SubElement ( parent, 'p' )
p1.text = ( 'Notes are grouped according to the reporting '
  'seasons for ' )
cite = et.SubElement ( p1, 'cite' )
cite.text = 'Audubon Field Notes'
cite.tail = '.'

#-- 2 --
# [ parent += paragraph linking to conventions page ]
p2 = et.SubElement ( parent, 'p' )
p2.text = 'See '
conLink = et.SubElement ( p2, 'a', href=CONVENTIONS_URL )
conLink.text = "How to read Shipman's field notes"
conLink.tail = ( ' for notational conventions and the '
  "author's contact information." )

```

## 16. indexTable(): Build the table of monthly links

For an overview of the structure of the index table, see Section 4.1, "XHTML for the index page" (p. 5).

noteweb

```

# - - -   i n d e x T a b l e

def indexTable ( parent, yearCollection ):
  '''Build the table of yearly rows and monthly link cells.

  [ (parent is an et.Element) and
    (yearCollection is a YearCollection instance) ->
    parent += the index table made from yearCollection ]
  ...

```

The `table` element, and everything in it down to but not including the `tbody` element, are built by Section 17, “`indexTableFrame()`: Set up the table structure” (p. 30).

noteweb

```
#-- 1 --
# [ parent += a new table element containing column
#       definitions, column headings, and an empty tbody
#       element
#   tbody := that tbody element ]
tbody = indexTableFrame ( parent )
```

All that remains is to fill in the rows of the table, one per year. The method described in Section 21.4, “`YearCollection.genYearsRev()`: Generate years in reverse chronological order” (p. 37) generates the years in reverse chronological order as a sequence of `YearRow` instances; each one results in one row of the table. See Section 18, “`buildRow()`: Build one row of the index table” (p. 31).

noteweb

```
#-- 2 -
# [ tbody += table rows made from the YearRow instances
#       in YearCollection, in reverse chronological order ]
for yearRow in yearCollection.genYearsRev():
  #-- 2 body --
  # [ yearRow is a YearRow instance ->
  #   tbody += a tr element made from yearRow ]
  buildRow ( tbody, yearRow )
```

## 17. `indexTableFrame()`: Set up the table structure

This function builds the index page's `table` element, the `col` elements describing the columns, the `thead` element labeling the columns, and the `tbody` element that will contain the yearly rows of the table.

noteweb

```
# - - -   i n d e x   T a b l e   F r a m e

def indexTableFrame ( parent ):
    '''Build an empty table.

    [ parent is an et.Element ->
      parent += a new table element containing column
                definitions, column headings, and an empty tbody
                element
      return that tbody element ]
    ...
```

For an overview of the XHTML, see Section 4.1, “XHTML for the index page” (p. 5).

noteweb

```
#-- 1 --
# [ parent += a new table element with class=SEASONS_CLASS
#   table := that element ]
table = et.SubElement ( parent, 'table', border='5',
                        cellpadding='5', width='100%' )
table.attrib['class'] = SEASONS_CLASS
```

```

#-- 2 --
# [ table += new col elements declaring the column
#           properties ]
et.SubElement ( table, 'colgroup', align='right' )
et.SubElement ( table, 'colgroup', span='12', align='left',
                width='*' )

#-- 3 --
# [ table += a thead element containing the column
#           labels ]
thead = et.SubElement ( table, 'thead' )
headRow = et.SubElement ( thead, 'tr' )
headRow.attrib['class'] = YEAR_ROW_CLASS
headLabel = et.SubElement ( headRow, 'th' )
headLabel.text = 'Year'

winter1 = et.SubElement ( headRow, 'th', colspan='2' )
winter1.attrib['class'] = 'winter'
winter1.text = 'Winter'

spring = et.SubElement ( headRow, 'th', colspan='3' )
spring.attrib['class'] = 'spring'
spring.text = 'Spring'

summer = et.SubElement ( headRow, 'th', colspan='2' )
summer.attrib['class'] = 'summer'
summer.text = 'Summer'

fall = et.SubElement ( headRow, 'th', colspan='4' )
fall.attrib['class'] = 'fall'
fall.text = 'Fall'

winter2 = et.SubElement ( headRow, 'th' )
winter2.attrib['class'] = 'winter'
winter2.text = 'Winter'

#-- 4 --
# [ table += a tbody element
#   tbody := that element ]
tbody = et.SubElement ( table, 'tbody' )

#-- 5 --
return tbody

```

## 18. buildRow(): Build one row of the index table

This function builds the row holding all the months in a single year of the index table.

noteweb

```

# - - -   b u i l d R o w

def buildRow ( tbody, yearRow ):
    '''Builds one row of the index table, holding one year.

```

```

    [ (tbody is an et.Element) and
      (yearRow is a YearRow instance) ->
        tbody += a tr element made from yearRow ]
    ...

```

The first step is to add the `tr` element to hold the table row. There is one minor complication: if the year number is divisible by the constant defined in Section 10.11, “YEAR\_GROUP\_FREQUENCY” (p. 21), we’ll add a `class='year-group'` attribute to produce a thicker lower border on that row, to improve legibility of the table.

noteweb

```

#-- 1 --
# [ tbody := a new 'tr' element
#   tr     := that element ]
tr = et.SubElement ( tbody, 'tr' )

#-- 2 --
# [ if int(yearRow.yyyy) is divisible by YEAR_GROUP_FREQUENCY ->
#   tr += a class=YEAR_GROUP_CLASS attribute
#   else ->
#   tr += a class=YEAR_ROW_CLASS attribute ]
if ( int(yearRow.yyyy) % YEAR_GROUP_FREQUENCY ) == 0:
    tr.attrib['class'] = YEAR_GROUP_CLASS
else:
    tr.attrib['class'] = YEAR_ROW_CLASS

```

The first cell in the row is always a `th` containing the row heading; it carries the CSS class name given in Section 10.14, “ROW\_LABEL\_CLASS” (p. 21), and is linked to the CSS rule in Section 5.6, “`th.row-label`: The row label” (p. 14).

noteweb

```

#-- 3 --
# [ tr += a th element with class=ROW_LABEL_CLASS
#   containing yearRow.yyyy ]
rowLabel = et.SubElement ( tr, 'th' )
rowLabel.attrib['class'] = ROW_LABEL_CLASS
rowLabel.text = yearRow.yyyy

```

We’ll need to add exactly twelve `td` elements, regardless of how many months have actual data (and it might even be that none of them do). Therefore, the loop that builds these cells is driven by iterating over the possible month keys ('01', '02', ..., '12'): for each key, we try to pull out the corresponding `MonthCell` from `yearRow`; this causes a `KeyError` if the year does not have that month. See Section 19, “`buildMonthCell()`: Build one monthly cell in the index table” (p. 33).

noteweb

```

#-- 4 --
# [ tr += twelve cells containing links to the elements
#   of yearRow corresponding to the twelve months, or
#   non-breaking spaces where months are absent ]
for monthKey in [ '%02d' % x
                  for x in range(1,13) ]:
    #-- 4 body --
    # [ monthKey is in the range ['01', '02', ..., '12'] ->
    #   if yearRow[monthKey] exists ->
    #     tr += a td element containing a link to

```

```

#           the month page for yearRow[monthKey]
#     else ->
#         tr += a td element containing a non-breaking
#             space ]
buildMonthCell ( tr, yearRow, monthKey )

```

## 19. buildMonthCell(): Build one monthly cell in the index table

This function builds one td cell in the index table.

noteweb

```

# - - -   b u i l d M o n t h C e l l

def buildMonthCell ( tr, yearRow, monthKey ):
    '''Add one month to the index table: a link or a non-breaking space.

    [ (tr is an et.Element) and
      (yearRow is a YearRow instance) and
      (monthKey is in the sequence '01', '02', ... '12') ->
        if yearRow[monthKey] exists ->
            tr += a td element containing a link to
                the month page for yearRow[monthKey]
        else ->
            tr += a td element containing a non-breaking
                space ]
    ...

```

Whether there is any month data or not, we must build a td element, and give it the CSS class name for the reporting season. For the mapping of month numbers to seasons, see Section 10.2, “MONTH\_SEASON\_MAP: Define the season for each month” (p. 19).

noteweb

```

#-- 1 --
# [ tr += a new td element
#   td := that element ]
td = et.SubElement ( tr, 'td' )

#-- 2 --
# [ td += a 'class' attribute equal to the season for
#     the month whose number is monthKey ]
td.attrib['class'] = MONTH_SEASON_MAP[monthKey]

```

Next, we check to see if there is any data for this month. If not, add the non-breaking space and we're done; see Section 10.15, “NBSP: Non-breaking space character” (p. 22).

noteweb

```

#-- 3 --
# [ if yearRow[monthKey] exists ->
#     monthCell := the corresponding value
#   else ->
#     td.text := a non-breaking space
#     return ]
try:

```

```

    monthCell = yearRow[monthKey]
except KeyError:
    td.text = NBSP
    return

```

Generate a link to the URL for the month page, computed by Section 23.2, “MonthCell.fileName(): Path to the month's page” (p. 48). For the mapping of month numbers to month names, see Section 10.1, “MONTH\_NAME\_MAP: Translate month numbers to month names” (p. 19).

noteweb

```

#-- 4 --
# [ td += a link to URL monthCell.fileName(), with the link
#       text the name month number (monthKey) ]
monthFileName = monthCell.fileName()
a = et.SubElement ( td, 'a', href=monthFileName )
a.text = "%s %s" % (monthKey, MONTH_NAME_MAP[monthKey])

```

## 20. class Args(): Command line argument processor

The purpose of this class is to check for valid command line arguments, and issue an error message otherwise. The class is a singleton: after its first instantiation, each call to the constructor returns the same instance. The Singleton base class is borrowed from the *Scanning objects for Python: Scan and Log*<sup>13</sup>.

noteweb

```

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

class Args(Singleton):
    '''Singleton class to represent command line arguments.

    Exports:
    Args():
        [ if the command line arguments are valid ->
          return the singleton Args() instance
          representing those arguments
        else ->
          sys.stderr += usage message
          stop execution ]
    .forceSwitch:
        [ if -f or --force was specified -> True
        else -> False ]
    ...

```

Overall processing and message generation is handled by Python's standard optparse module<sup>14</sup>.

noteweb

```

def __init__(self):
    '''Constructor.
    ...

#-- 1 --
# [ parser := a new optparse.OptionParser instance

```

<sup>13</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/>

<sup>14</sup> <http://docs.python.org/library/optparse.html>

```

#     representing the valid switches ]
parser = optparse.OptionParser ( version="%s %s" %
    (PROGRAM_NAME, EXTERNAL_VERSION) )
parser.add_option ( "-f", "--force",
    dest="force", default=False, action="store_true",
    help=("Rewrite all HTML files, not just those that are "
        "out of date.") )

#-- 2 --
# [ if (parser likes the command line arguments) and
#   (there are no positional arguments) ->
#   options := switch options
#   argList := positional arguments
#   else ->
#   sys.stderr += usage message
#   stop execution ]
options, argList = parser.parse_args()
if len(argList) > 0:
    parser.error ( "This script takes no positional arguments." )

```

Finally, copy the value of the `.force` attribute of the returned `options` instance to the exported `.forceSwitch` attribute.

noteweb

```

#-- 3 --
self.forceSwitch = options.force

```

## 21. class YearCollection: The top-level data structure

The single instance of this class holds the entire collection of years and months.

noteweb

```

# - - - - - c l a s s   Y e a r C o l l e c t i o n

class YearCollection:
    '''Represents the entire set of years and months.

    Exports:
    YearCollection():
        [ return a new, empty YearCollection instance ]
    .addYear ( yyyy ):
        [ yyyy is a year number as a four-digit string ->
          self := self with a new, empty YearRow
              added for year=yyyy
          return that YearRow ]
    .__getitem__(self, yyyy):
        [ yyyy is a year number as a four-digit string ->
          if yyyy is contained in self ->
              return the corresponding YearRow instance
          else -> raise KeyError ]
    .genYearsRev():
        [ generate self's contained YearRow instances in
          reverse chronological order ]

```

```

.neighbors ( yyyy_mm ):
    [ yyyy_mm is a month string as 'yyyy-mm' ->
      let:
          prev == the month in self prior to yyyy_mm in
                  chronological sequence as a 'yyyy-mm' string,
                  or None if first
          next == the month in self after yyyy_mm in
                  chronological sequence as a 'yyyy-mm' string,
                  or None if last
      in:
          return (prev,next) ]
.findMonth ( yyyy_mm ):
    [ yyyy_mm is a month key string as 'yyyy-mm' ->
      if self has a month with that key ->
          return the corresponding MonthCell
      else -> raise KeyError ]

```

Inside the instance, we will need someplace to store the contained YearRow instances. Because the set of years might have gaps in it, we can use a dictionary whose keys are year numbers.

noteweb

```

State/Invariants:
    .__yearMap:
        [ a dictionary whose keys are the 'yyyy' strings
          of years in self, and each corresponding value
          is a YearRow instance for that year ]
    ...

```

## 21.1. YearCollection.\_\_init\_\_(): Constructor

All the constructor needs is to set up the invariant on the `.__yearMap` attribute by making it an empty dictionary.

noteweb

```

# - - -   Y e a r C o l l e c t i o n . _ _ i n i t _ _

def __init__ ( self ):
    '''Constructor for YearCollection.
    ...
    self.__yearMap = {}

```

## 21.2. YearCollection.addYear(): Add a year

This method creates a new YearRow instance and adds it to self. See Section 22, “class YearRow: Container for one year's records” (p. 41).

noteweb

```

# - - -   Y e a r C o l l e c t i o n . a d d Y e a r

def addYear ( self, txny, yyyy ):
    '''Add a new year row
    ...
    yearRow = YearRow ( self, txny, yyyy )

```

```
self.__yearMap[yyyy] = yearRow
return yearRow
```

### 21.3. YearCollection.\_\_getitem\_\_(): Return self[yyyy]

This method implements the index ( [... ] ) operation on a YearCollection. If there is no such year, it will raise KeyError.

noteweb

```
# - - -   Y e a r C o l l e c t i o n . _ _ g e t i t e m _ _
def __getitem__ ( self, yyyy ) :
    '''Implements self[yyyy]
    ...
    return self.__yearMap[yyyy]
```

### 21.4. YearCollection.genYearsRev(): Generate years in reverse chronological order

noteweb

```
# - - -   Y e a r C o l l e c t i o n . g e n Y e a r s R e v
def genYearsRev ( self ) :
    '''Generate years in reverse chronological order
    ...
```

The set of keys to self.\_\_yearMap is the set of years. We extract it, sort it, reverse it, and then generate the contained YearRow instances in that order.

noteweb

```
#-- 1 --
# [ yyyyList := keys of self.__yearMap in descending
#           order ]
yyyyList = self.__yearMap.keys()
yyyyList.sort()
yyyyList.reverse()

#-- 2 --
# [ generate the YearRow instances in self in order by
#   the elements of yyyyList ]
for yyyy in yyyyList:
    yield self[yyyy]

#-- 3 --
raise StopIteration
```

### 21.5. YearCollection.neighbors(): Find previous and next month

For a given month key expressed as a 'yyyy-mm' string, this method finds the preceding and following months, and returns a two-element tuple containing the month keys of its immediate neighbors, using None for the predecessor of the first month in the set, or for the successor of the last month.

```
# - - - YearCollection.neighbors

def neighbors ( self, yyyy_mm ):
    '''Find the previous/next months to yyyy_mm in sequence
    ...
```

Because we use a two-level structure of years and months, and because the predecessor or successor of a month may be in a different year, we'll use this algorithm to find the predecessor.

1. If year `yyyy` has any months before `mm`, the predecessor is the last such month.
2. Work backwards through years `yyyy-1`, `yyyy-2`, until we find a year that has at least one month in it. When we do, the predecessor is the last month in that year.
3. If we run out of years, the predecessor is `None`.

The algorithm to find the successor is symmetric. See Section 21.6, “`YearCollection.__findPrev()`: Find predecessor month” (p. 38) and Section 21.7, “`YearCollection.findNext()`: Find successor month” (p. 40).

```
#- 1 -
# [ if yyyy_mm has a predecessor in self ->
#   prev := that predecessor's 'yyyy-mm' month key
#   else ->
#     prev := None ]
prev = self.__findPrev ( yyyy_mm )

#- 2 -
# [ if yyyy_mm has a successor in self ->
#   next := that successor's 'yyyy-mm' month key
#   else ->
#     next := None ]
next = self.__findNext ( yyyy_mm )

#- 3 -
return (prev, next )
```

## 21.6. `YearCollection.__findPrev()`: Find predecessor month

For the algorithm, see Section 21.5, “`YearCollection.neighbors()`: Find previous and next month” (p. 37).

```
# - - - YearCollection.__findPrev

def __findPrev ( self, yyyy_mm ):
    '''Find the month preceding yyyy_mm.

    [ yyyy_mm is a month in self with key string 'yyyy-mm' ->
      if self has any months before yyyy_mm ->
        return the month key of the immediate predecessor
      else -> return None ]
    ...
```

First we find the YearRow containing yyyy\_mm, and ask that instance if the month has a predecessor in the same year.

noteweb

```
#-- 1 --
# [ yyyy := year part of yyyy_mm
#   mm   := month part of yyyy_mm ]
yyyy, mm = yyyy_mm.split('-')

#-- 2 --
# [ yyyy is a year in self ->
#   yearRow := YearRow instance for year yyyy ]
yearRow = self[yyyy]

#-- 3 --
# [ if yearRow has a predecessor to month mm ->
#   return that month's key as a string 'mm'
#   else -> I ]
try:
    prev = yearRow.predecessor ( mm )
    return '%s-%s' % (yyyy, prev)
except KeyError:
    pass
```

If there is a predecessor, it must be in a previous year. First we form a list of the year numbers in descending order. Then we find the position of yyyy in that list, and step through the remaining elements (if any) until we find a year that has at least one month in it, and return the last month.

noteweb

```
#-- 4 --
# [ yyyyList := all year key strings in self, sorted
#             in descending order ]
yyyyList = self.__yearMap.keys()
yyyyList.sort()
yyyyList.reverse()

#-- 5 --
# [ yyyy is an element of yyyyList ->
#   pos := yyyy's position in yyyyList ]
pos = yyyyList.index ( yyyy )
```

For the method that looks for the last month of a year, see Section 22.5, “YearRow.lastMonth(): Return the last month” (p. 43).

noteweb

```
#-- 6 --
# [ if any year with a key in yyyyList[pos+1:] has at
#   least one month in it ->
#   return the yyyy-mm key of the last month in the
#   first such year
#   else -> I ]
for prevYear in yyyyList[pos+1:]:
    prevRow = self[prevYear]
    if len(prevRow) > 0:
        lastMM = prevRow.lastMonth()
        return '%s-%s' % (prevYear, lastMM)
```

```
#-- 7 --  
return None
```

## 21.7. YearCollection.findNext(): Find successor month

This method is the mirror image of Section 21.6, “YearCollection.\_\_findPrev(): Find predecessor month” (p. 38). The algorithm is discussed in Section 21.5, “YearCollection.neighbors(): Find previous and next month” (p. 37).

noteweb

```
# - - -   Y e a r C o l l e c t i o n . _ _ f i n d N e x t  
  
def __findNext ( self, yyyy_mm ):  
    '''Find the next month chronologically after yyyy_mm.  
  
    [ yyyy_mm is a month in self with key string 'yyyy-mm' ->  
      if self has any months after yyyy_mm ->  
        return the month key of the immediate successor  
      else -> return None ]  
    ...  
#-- 1 --  
# [ yyyy  := year part of yyyy_mm  
#   mm    := month part of yyyy_mm ]  
yyyy, mm = yyyy_mm.split('-')  
  
#-- 2 --  
# [ yyyy is a year in self ->  
#   yearRow := YearRow instance for year yyyy ]  
yearRow = self[yyyy]  
  
#-- 3 --  
# [ if yearRow has a successor to month mm ->  
#   return that month's key as a string 'mm'  
#   else -> I ]  
try:  
    next = yearRow.successor ( mm )  
    return '%s-%s' % (yyyy, next)  
except KeyError:  
    pass  
  
#-- 4 --  
# [ yyyyList := all year key strings in self, sorted  
#             in ascending order ]  
yyyyList = self.__yearMap.keys()  
yyyyList.sort()  
  
#-- 5 --  
# [ yyyy is an element of yyyyList ->  
#   pos := yyyy's position in yyyyList ]  
pos = yyyyList.index ( yyyy )
```

For the function that returns the first month of a given year, see Section 22.4, “YearRow.firstMonth(): Return the first month” (p. 43).

noteweb

```
#-- 6 --
# [ if any year with a key in yyyyList[pos+1:] has at
#   least one month in it ->
#   return the yyyy-mm key of the first month in
#   the first such year
# else -> I ]
for nextYear in yyyyList[pos+1:]:
    nextRow = self[nextYear]
    if len(nextRow) > 0:
        firstMM = nextRow.firstMonth()
        return '%s-%s' % (nextYear, firstMM)

#-- 7 --
return None
```

## 22. class YearRow: Container for one year's records

Each instance of this class holds the information in one row of the index table: the year number, and a list of the months for which valid HTML monthly pages have been built.

noteweb

```
# - - - - - c l a s s   Y e a r R o w

class YearRow:
    '''Represents one year's line in the index table.

    Exports:
    YearRow ( yearCollection, txny, yyyy ):
        [ (yearCollection is the parent YearCollection) and
          (txny is a taxonomy.Txny instance) and
          (yyyy is a four-digit year as a string) ->
          return a new, empty YearRow with year=yyyy ]
    .yearCollection: [ as passed to constructor, read-only ]
    .txny:           [ as passed to constructor, read-only ]
    .yyyy:           [ as passed to constructor, read-only ]
    .__len__(self):
        [ return the number of months in self ]
    .__getitem__(self, mm):
        [ mm is a month key as 'mm' ->
          if self has a month mm ->
            return the corresponding MonthCell
          else -> raise KeyError ]
    .firstMonth():
        [ if self has any months ->
          return the month key of the first as 'mm'
          else -> raise KeyError ]
    .lastMonth():
        [ if self has any months ->
```

```

        return the month key of the last as 'mm'
        else -> raise KeyError ]
    .predecessor ( mm ):
        [ if 'mm' is a month key in self ->
          if self has any months before mm ->
            return the last such month key as 'mm'
          else -> raise KeyError ]
    .successor ( mm ):
        [ if 'mm' is a month key in self ->
          if self has any months after mm ->
            return the first such month key as 'mm'
          else -> raise KeyError ]
    .readAllMonths():
        [ self := self with MonthCell instances added for
          XML notes files in directory yyyy valid
          against birdnotes.rnc and txny
          sys.stderr += error message(s) for invalid XML
          notes files in that directory, if any ]

State/Invariants:
    .__monthMap:
        [ a dictionary whose keys are two-digit, zero-filled
          month numbers, and each corresponding value is the
          MonthCell instance representing that month's
          field notes ]
    ...

```

## 22.1. YearRow.\_\_init\_\_(): Constructor

The constructor has little to do: save the year number, and set up an empty \_\_monthMap.

noteweb

```

# - - -   Y e a r R o w .   _ _   i n i t   _ _

def __init__ ( self, yearCollection, txny, yyyy ):
    '''Constructor
    ...

    self.yearCollection = yearCollection
    self.txny = txny
    self.yyyy = yyyy
    self.__monthMap = {}

```

## 22.2. YearRow.\_\_len\_\_(): Number of contained months

This method returns the number of months in self. Note: if there is a year directory but there were no valid month files in it, this method will return zero.

noteweb

```

# - - -   Y e a r R o w .   _ _   l e n   _ _

def __len__ ( self ):
    '''Return the number of contained months.

```

```
...
return len(self.__monthMap)
```

### 22.3. YearRow.\_\_getitem\_\_(): Retrieve one month

noteweb

```
# - - -   Y e a r R o w . _ _ g e t i t e m _ _
def __getitem__( self, mm ):
    '''Retrieve one month
    ...
    return self.__monthMap[mm]
```

### 22.4. YearRow.firstMonth(): Return the first month

noteweb

```
# - - -   Y e a r R o w . f i r s t M o n t h
def firstMonth(self):
    '''Retrieve self's first month number, if any.
    ...
    #-- 1 --
    if len(self) == 0:
        raise KeyError

    #-- 2 --
    # [ monthKeyList := month keys in self in ascending order ]
    monthKeyList = self.__monthMap.keys()
    monthKeyList.sort()

    #-- 3 --
    # [ return the month number from the first element
    #   of monthKeyList ]
    return self[monthKeyList[0]].mm
```

### 22.5. YearRow.lastMonth(): Return the last month

noteweb

```
# - - -   Y e a r R o w . l a s t M o n t h
def lastMonth(self):
    '''Retrieve self's last month, if any.
    ...
    #-- 1 --
    if len(self) == 0:
        raise KeyError

    #-- 2 --
    # [ monthKeyList := month keys in self in ascending order ]
    monthKeyList = self.__monthMap.keys()
    monthKeyList.sort()
```

```

#-- 3 --
# [ return the month number from the last element
#   of monthKeyList ]
return self[monthKeyList[-1]].mm

```

## 22.6. YearRow.predecessor(): Find the previous month

This method finds the month that precedes a given month, if there is one.

noteweb

```

# - - -   Y e a r R o w . p r e d e c e s s o r

def predecessor ( self, mm ):
    '''Find the month preceding month mm, if any
    ...

```

First we find the set of month keys in `self`, then sort it in descending order, so that the preceding month's key (if any) will follow `mm`'s position in that list. We get to assume that `mm` is a member of the list, by the precondition on this method.

noteweb

```

#-- 1 --
# [ mmList := keys of self.__monthMap in descending order ]
mmList = self.__monthMap.keys()
mmList.sort()
mmList.reverse()

#-- 2 --
# [ mm is a member of mmList ->
#   pos := position of mm in mmList ]
pos = mmList.index ( mm )

#-- 3 --
# [ if pos+1 >= len(mmList) ->
#   raise KeyError
#   else ->
#   return mmList[pos+1] ]
if pos+1 >= len(mmList):
    raise KeyError
else:
    return mmList[pos+1]

```

## 22.7. YearRow.successor(): Find the following month

This method is the mirror image of Section 22.6, “YearRow.predecessor(): Find the previous month” (p. 44). The only difference is that we sort the key list in ascending order instead of descending.

noteweb

```

# - - -   Y e a r R o w . s u c c e s s o r

def successor ( self, mm ):
    '''Find the following month.
    ...

```

```

#-- 1 --
# [ mmList := keys of self.__monthMap in ascending order ]
mmList = self.__monthMap.keys()
mmList.sort()

#-- 2 --
# [ mm is a member of mmList ->
#   pos := position of mm in mmList ]
pos = mmList.index ( mm )

#-- 3 --
# [ if pos+1 >= len(mmList) ->
#   raise KeyError
#   else ->
#   return mmList[pos+1] ]
if pos+1 >= len(mmList):
    raise KeyError
else:
    return mmList[pos+1]

```

## 22.8. YearRow.readAllMonths(): Process input files for one year

This method looks in subdirectory `self.yyyy` to see if there are any files whose names look like monthly XML input files, and attempts to read them all. Those which are valid will be stored in `self.__monthMap`; the invalid ones will generate error messages.

noteweb

```

# - - -   Y e a r R o w . r e a d A l l M o n t h s

def readAllMonths ( self ):
    '''Find and read all monthly files for this year.
    ...

```

We use `os.listdir()` to get a list of all the names in the year directory, filtering it with the regular expression declared in Section 10.4, “YYYY\_MM\_XML\_PAT: Month file name pattern” (p. 20).

noteweb

```

#-- 1 --
# [ monthFileNameList := names from directory
#   ('./'+self.yyyy) that match YYYY_MM_XML_PAT ]
monthFileNameList = [ fileName
    for fileName in os.listdir ( self.yyyy )
    if YYYY_MM_XML_PAT.match(fileName) is not None ]

```

For each month file, we attempt to open and read that file using the `birdnotes.BirdNoteSet` constructor. The `txny` argument is necessary to define the valid bird codes. See Section 22.9, “YearRow.readOneMonth(): Read one monthly file” (p. 46).

noteweb

```

#-- 2 --
# [ self := self with MonthCell instances added
#   representing files in monthFileNameList
#   valid against birdnotes.rnc and txny
#   sys.stderr += error message(s) for invalid
#   files, if any ]

```

```

for monthFileName in monthFileNameList:
    #-- 2 body --
    # [ if monthFileName names a readable file
    #   valid against birdnotes.rnc and txny ->
    #     self.__monthMap += an entry whose key is the
    #                       month number and whose value is a MonthCell
    #                       representing that file
    #   else ->
    #     sys.stderr += error message(s) ]
    self.readOneMonth ( monthFileName )

```

## 22.9. YearRow.readOneMonth(): Read one monthly file

This method attempts to read one monthly XML input file.

noteweb

```

# - - -   Y e a r R o w . r e a d O n e M o n t h

def readOneMonth ( self, monthFileName ):
    '''Try to read one month's field data.

    [ monthFileName is a string matching YYYY_MM_XML_PAT ->
      if monthFileName names a readable file
      valid against birdnotes.rnc and txny ->
        self.__monthMap += an entry whose key is the
                          month number and whose value is a MonthCell
                          representing that file
      else ->
        sys.stderr += error message(s) ]
    ...

```

Since our precondition guarantees that `monthFileName` has the form `'yyyy-mm.xml'`, we can extract the month name by simple slicing.

noteweb

```

#-- 1 --
# [ mm := month number from monthFileName
#   monthPath := self.yyyy + '/' + monthFileName ]
#--
# 0      1
# 01234567890
# yyyy-mm.xml
#--
mm = monthFileName[5:7]
monthPath = '%s/%s' % (self.yyyy, monthFileName)

```

We'll create a `BirdNoteSet` instance whose taxonomy comes from our `self.txny` attribute. Then we'll call its `.readFile()` method to add the month file (if valid).

noteweb

```

#-- 2 --
# [ birdNoteSet := a new birdnotes.BirdNoteSet instance
#   with taxonomy self.txny ]
birdNoteSet = birdnotes.BirdNoteSet ( self.txny )

```

```

#-- 3 --
# [ if monthPath names a readable file valid against
#   birdnotes.rnc ->
#     birdNoteSet := birdNoteSet with data added
#     from that file
#   else ->
#     sys.stderr += error message(s)
#     return ]
try:
    birdNoteSet.readFile ( monthPath )
except IOError, detail:
    print >>sys.stderr, ( "*** Invalid monthly file "
        "'%s': %s" % (monthFileName, detail) )
    return

```

Finally, create a new `MonthCell` instance and add it to `self.__monthMap`. See Section 23, “class `MonthCell`: One table cell” (p. 47).

noteweb

```

#-- 4 --
# [ self.__monthMap[mm] := a new MonthCell instance
#   with self as the parent, month=mm, and
#   birdNoteSet=birdNoteSet ]
self.__monthMap[mm] = MonthCell ( self, mm, birdNoteSet )

```

## 22.10. YearRow.writeMonthPages(): Generate monthly pages

This method looks in the year directory for each XML file that contains a month's worth of notes. It translates the valid ones to HTML and updates the list of months.

noteweb

```

# - - -   Y e a r R o w . w r i t e M o n t h P a g e s

def writeMonthPages ( self ):
    '''Generate all monthly pages for this year.

    [ monthly web pages := rendering of valid monthly XML
      files under directory self.year, if any ]
    ...

```

The `self.__monthMap` dictionary contains month keys of the form `'mm'`, with each value a `MonthCell` instance that knows how to translate its monthly page. See Section 23.3, “`MonthCell.writePage()`: Render as XHTML” (p. 49).

noteweb

```

#-- 1 --
for mm in self.__monthMap:
    monthCell = self.__monthMap[mm]
    monthCell.writePage()

```

## 23. class MonthCell: One table cell

Each instance of this class holds all the information we need to build one month's cell in the index table.

```
# - - - - - c l a s s   M o n t h C e l l

class MonthCell:
    '''Represents one month's entry in the index table.

    Exports:
    MonthCell ( yearRow, mm, birdNoteSet ):
        [ (yearRow is the containing YearRow instance) and
          (mm is the month as a left-zero-filled string 'mm') and
          (birdNoteSet is a birdnotes.BirdNoteSet instance
           representing that month's data) ->
          return a new MonthCell instance with those values ]
    .yearRow:      [ as passed to constructor, read-only ]
    .mm:           [ as passed to constructor, read-only ]
    .yyyy_mm:     [ (yearRow.yyyy)+'-'+(mm)
    .title:       [ self's 'monthName yyyy' string ]
    .birdNoteSet: [ as passed to constructor, read-only ]
    .fileName():
        [ return the path name of the month's file relative
          to the index page ]
    .writePage():
        [ if self's XHTML page can be created anew ->
          that page := self rendered as XHTML ]
    ...
```

### 23.1. MonthCell.\_\_init\_\_(): Constructor

```
# - - -   M o n t h C e l l .   _ _ i n i t _ _

def __init__ ( self, yearRow, mm, birdNoteSet ):
    '''Constructor for MonthCell.
    ...

    self.yearRow      = yearRow
    self.mm           = mm
    self.yyyy_mm      = '%s-%s' % (yearRow.yyyy, mm)
    self.birdNoteSet  = birdNoteSet
    self.title        = ( "Shipman's field notes, %s" %
                          MonthCell.monthName ( self.yyyy_mm ) )
```

### 23.2. MonthCell.fileName(): Path to the month's page

This method returns the name of the monthly report file. The subdirectory is the name of the year, and the file's name is `self.yyyy_mm` plus the standard HTML extension.

```
# - - -   M o n t h C e l l .   f i l e N a m e

def fileName ( self ):
    '''Return this month's page's relative path.
```

```

...
return ( '%s/%s%s' %
        (self.yearRow.yyyy, self.yyyy_mm, HTML_EXT) )

```

### 23.3. MonthCell.writePage(): Render as XHTML

This method creates the page containing one month's notes, provided the page is out of date.

noteweb

```

# - - - M o n t h C e l l . w r i t e P a g e

def writePage ( self ):
    '''Render self as XHTML.
    ...

```

We don't want to rewrite every HTML file every time, so we use the `self.birdNoteSet.newestTime` timestamp to see if all of the source files are older than the output file and, if so, return.

noteweb

```

#-- 1 --
# [ if (not Args().forceSwitch) and
#   (self.fileName() names an existing file whose modification
#   time is greater than self.birdNoteSet.newestTime) ->
#   return
#   else ->
#     outFile_name := self.fileName() ]
outFile_name = self.fileName()
if not Args().forceSwitch:
    if os.path.exists ( outFile_name ):
        status = os.stat ( outFile_name )
        modTime = status[stat.ST_MTIME]
        if ( (self.birdNoteSet.newestTime is not None) and
            (modTime > self.birdNoteSet.newestTime) ):
            return

```

Before we can create the page, we need to find the previous and next page so this page's *Previous* and *Next* links can point to them.

noteweb

```

#-- 2 --
# [ prev := the 'yyyy-mm' key of the preceding month's
#         page, or None if self is the first month
#   next := the 'yyyy-mm' key of the following month's
#         page, or None if self is the last month ]
prev, next = self.yearRow.yearCollection.neighbors (
    self.yyyy_mm )

```

At this point we call the `tccpage2.TCCPage()` constructor to set up a basic page structure. See Section 23.4, “MonthCell.\_\_pageFrame(): Set up a basic page” (p. 50).

noteweb

```

#-- 3 --
# [ page := a tccpage2.TCCPage instance with
#         navigation links (prev, next) ]
page = self.__pageFrame ( prev, next )

```

For the method that generates the page's variable content, see Section 23.5, “MonthCell.\_\_renderPage()”: Add the notes content” (p. 52).

noteweb

```
#-- 4 --
# [ page := page with self.birdNoteSet rendered as XHTML ]
self.__renderPage(page)
```

Open the page's file for writing (if possible) and write the content there.

noteweb

```
#-- 5 --
# [ if a file named self.fileName() can be opened new ->
#   outFile := that file, so opened
#   else ->
#     sys.stderr += error message
#     stop execution ]
try:
    outFile = open ( outFile_name, 'w' )
except IOError, detail:
    print >>sys.stderr, ( "*** Can't create monthly page "
        "'%s': %s" % (outFileName, detail) )
    raise SystemExit

#-- 6 --
# [ outFile += page, rendered as XHTML ]
page.write ( outFile )
outFile.close()
```

## 23.4. MonthCell.\_\_pageFrame(): Set up a basic page

This method sets up a TCCPage instance to hold the month's content. Links to the documentation for `tccpage2.py` are in Section 3, “Overview of the internals” (p. 4).

noteweb

```
# - - - M o n t h C e l l . _ _ p a g e F r a m e

def __pageFrame ( self, prev, next ):
    '''Set up an empty tccpage2.TCCPage instance.

    [ (prev is the 'yyyy-mm' of the previous month or None) and
    [ (next is the 'yyyy-mm' of the next month or None) ->
      return a new TCCPage instance with navigation links
      previous=(prev's page) and next=(next's page) ]
    ...
```

For the navigational link plan, see Section 4, “The generated XHTML” (p. 5). We have four navigational features: *Next* links to `nextURL`; *Previous* links to `prevURL`; *Contents* goes to the index page; and *Home* goes to Shipman's home page.

To transform the target month numbers into relative URLs, we can't assume that the referenced pages are in the same directory. For instance, the predecessor of `'1998-01.xml'` might be URL `'../1997/1997-12.xml'`. For simplicity, we'll always go up one level and then back down.

```

#-- 1 --
# [ if prev is None ->
#   prevList := an empty list
#   else ->
#   prevList := [(name of month prev,URL of prev)] ]
#--
# 0123456
# yyyy-mm <-- Value of 'prev' or 'next'
#--
if prev is None:
    prevList = []
else:
    prevYYYY = prev[:4]
    prevMM   = prev[5:]
    prevURL  = ( '../%s/%s%s' %
                (prevYYYY, prev, HTML_EXT) )
    prevText = MonthCell.monthName(prev)
    prevList = [(prevText, prevURL)]

#-- 2 --
# [ if next is None ->
#   nextList := an empty list
#   else ->
#   nextList := [(name of month next,URL of next)] ]
if next is None:
    nextList = []
else:
    nextYYYY = next[:4]
    nextMM   = next[5:]
    nextURL  = ( '../%s/%s%s' %
                (nextYYYY, next, HTML_EXT) )
    nextText = MonthCell.monthName(next)
    nextList = [(nextText, nextURL)]

```

Next, build the list of NavLink instances describing the page's navigational features.

```

#-- 2 --
# [ navList := a list of tccpage2.NavLink instances
#   representing the navigational features ]
navList = (
    tccpage2.NavLink ( 'Next', nextList ),
    tccpage2.NavLink ( 'Previous', prevList ),
    tccpage2.NavLink ( 'Contents',
        ["Shipman's Field Notes",
         '../%s%s' % (INDEX_PAGE_NAME, HTML_EXT)] ),
    tccpage2.NavLink ( 'Home',
        ["Shipman's Home Sweet Homepage", HOME_PAGE_URL] ) )

```

All that remains is to build and return the TCCPage instance.

```
#-- 3 --
return tccpage2.TCCPage ( self.title, navList,
                          logoImage=ZDP_LOGO, logoLink=ZDP_URL, cssUrl=CSS_URL )
```

## 23.5. MonthCell.\_\_renderPage(): Add the notes content

Translate `self.birdNoteSet` to XHTML. The `page` argument is a `tccpage2.TCCPage` instance whose `.content` attribute is an element under which all the page content is placed. For an outline of the XHTML at this level, see Section 4.2, “XHTML for the month page” (p. 6).

The page starts with a table of links to the daily data blocks, followed by one data block for each day. This rendering is driven by the `self.birdNoteSet.genDays()` method, which generates the daily blocks as a sequence `birdnotes.DayNotes` instances, in the order they occur in the input.

This process is complicated by the need to set up a system of anchor names so that the page's table of contents (TOC) can link correctly to the rendering of each `DayNotes` instance in the current month.

Because there may be more than one `day-notes` element with the same date, we can't simply use the “`yyyy-dd-mm`” date strings as anchor names. We'll have to add a suffix to separate them, e.g., “`2008-04-11`”, “`2008-04-11a`”, “`2008-04-11b`”, and so forth.

We'll need two local data structures to manage this process:

- A Python `set` instance named `anchorSet` will accumulate the anchor strings that we have used so far. This is needed only during the process of assigning unique anchor values, which occurs in Section 23.6, “`MonthCell.__pageTOC(): Generate page table of contents`” (p. 53), so it is local to that method.
- In order to remember which anchors correspond to which `DayNotes` instances, we'll use a rather interesting little Python trick. In Python, you can use an instance as an index in a dictionary. We'll keep the anchors as values in a dictionary named `anchorMap`, and the keys will be the `DayNotes` instances in this month. That structure is returned by the `.__pageTOC` method.

noteweb

```
# - - - M o n t h C e l l . r e n d e r P a g e

def __renderPage ( self, page ):
    '''Render the notes into XHTML.

    [ page is a tccpage2.TCCPage instance ->
      page := page with self.birdNoteSet rendered as XHTML ]
    ...
```

First we add the short paragraph linking to the documentation so readers can interpret the notes.

noteweb

```
#-- 1 --
# [ page.content += paragraph linking to documentation ]
intro = et.SubElement ( page.content, 'p' )
introLink = et.SubElement ( intro, 'a',
                           href=CONVENTIONS_URL )
introLink.text = "How to read Shipman's field notes"
```

Next comes the page table of contents. See Section 23.6, “`MonthCell.__pageTOC(): Generate page table of contents`” (p. 53).

```

#-- 2 --
# [ page.content += a 'ul' containing the page table
#   of contents (TOC)
#   anchorMap += a dictionary whose keys are the DayNotes
#   instances in self.birdNoteSet, and each
#   corresponding value is the anchor name used for
#   that instance in the generated TOC ]
anchorMap = self.__pageTOC ( page.content )

```

Finally, we generate a block for each day-notes element in the month. See Section 23.10, “MonthCell.\_\_dayBlock(): Render one daily note set” (p. 57).

```

#-- 3 --
# [ page.content += XHTML rendering of the day-notes
#   elements inside self.birdNoteSet, using anchorMap
#   for the mapping of those elements onto anchors ]
for dayNotes in self.birdNoteSet.genDays():
  #-- 3 body --
  # [ dayNotes is a birdnotes.DayNotes instance ->
  #   page.content += XHTML rendering of dayNotes
  #   defining anchor anchorMap[dayNotes] ]
  self.__dayBlock ( page.content, dayNotes,
                    anchorMap[dayNotes] )

```

## 23.6. MonthCell.\_\_pageTOC(): Generate page table of contents

This method generates the table of contents for one monthly page. For the XHTML generated, see Section 4.2, “XHTML for the month page” (p. 6).

```

# - - -   M o n t h C e l l . _ _ p a g e T O C

def __pageTOC ( self, parent ):
    '''Generate the monthly page's table of contents.

    [ parent is an et.Element ->
      parent += a 'ul' containing the page table
              of contents
      return a dictionary whose keys are the DayNotes
      instances in self.birdNoteSet, and each
      corresponding value is the anchor name used for
      that instance in the generated TOC ]
    ...

```

At the top level, a ul (bullet list) wraps the table of contents. Also, create the anchorSet and anchorMap structures discussed under Section 23.5, “MonthCell.\_\_renderPage(): Add the notes content” (p. 52).

```

#-- 1 --
# [ parent := parent with a new 'ul' child element added
#   ul := that child
#   anchorSet := a new, empty set
#   anchorMap := a new, empty dictionary ]

```

```

ul = et.SubElement ( parent, 'ul' )
anchorSet = set()
anchorMap = {}

```

For the logic that generates each TOC entry, see Section 23.7, “MonthCell.\_\_dayTOC(): Month table of contents entry for one day” (p. 54).

noteweb

```

#-- 2 --
# [ ul += 'li' elements representing the days in
#     self.birdNoteSet
#   anchorSet += anchors used in the table of contents
#   anchorMap += entries mapping the DayNotes
#     instances in self.birdNoteSet onto the corresponding
#     anchors ]
for dayNotes in self.birdNoteSet.genDays():
    #-- 2 body --
    # [ ul += an 'li' element representing dayNotes ]
    self.__dayTOC ( ul, dayNotes, anchorSet, anchorMap )

```

The finished anchorMap is returned to the caller.

noteweb

```

#-- 3 --
return anchorMap

```

## 23.7. MonthCell.\_\_dayTOC(): Month table of contents entry for one day

This method generates one entry in the table of contents at the top of the monthly page.

noteweb

```

# - - -   M o n t h C e l l . _ _ d a y T O C

def __dayTOC ( self, ul, dayNotes, anchorSet, anchorMap ):
    '''Generate a link to the day's entry, with notables if any.

    [ (ul is an et.Element) and
      (dayNotes is a DayNotes instance) and
      (anchorSet is a set) and
      (anchorMap is a dictionary) ->
        ul += an 'li' item containing a link to the
            anchor for dayNotes.date, plus the English
            names for any notable records
        anchorSet += a new anchor value not in anchorSet
        anchorMap += an entry whose key is dayNotes
                    and whose value is that new anchor value ]
    ...

```

First we must create an anchor string that hasn't been used yet. We'll start with "D" plus the `dayNotes.date` string (which has the form “yyyy-mm-dd”). The `anchorSet` argument is a Python `set` instance containing all the anchors that have already been used, so if the simple date is not in that set, we'll use that. If not, we'll try suffix letters 'a', 'b', and so on, until we get a new one.

In theory, if there are 27 or more daily sets, we might run out of letters. However, the author rarely records multiple sets per day. The worst case is a long drive through multiple states, but to drive through 27 states in one day would be a pretty good trick.

noteweb

```
#-- 1 --
# [ newAnchor := a valid HTML anchor name that is not in
#       anchorSet ]
newAnchor = "D" + dayNotes.date
suffix = 'a'
while newAnchor in anchorSet:
    newAnchor = "D" + dayNotes.date+suffix
    suffix = chr(ord(suffix)+1)
```

Now that we have an anchor string `newAnchor` that isn't in `anchorSet`, we know we can use it as the anchor for `dayNotes`.

noteweb

```
#-- 2 --
# [ anchorSet := union(anchorSet, newAnchor)
#       anchorMap += an entry whose key is dayNotes and
#       whose value is newAnchor ]
anchorSet.add ( newAnchor )
anchorMap[dayNotes] = newAnchor
```

Next we'll create the `li` (bullet) for this date. Inside the bullet is a link to the anchor we just generated, whose link text is the date, state, and day locality string from `dayNotes`.

noteweb

```
#-- 3 --
# [ ul += a new 'li' element
#       li += that element ]
li = et.SubElement ( ul, 'li' )

#-- 4 --
# [ li += a new 'a' element whose URL is ('#+newAnchor)
#       and whose link text is (dayNotes.date)+' : '+'
#       (dayNotes.regionCode, uppercased)+' : '+'
#       (dayNotes.dayLoc.name) ]
a = et.SubElement ( li, 'a', href=('#+newAnchor') )
a.text = dayNotes.title()
```

Next we must scan through all the records inside `dayNotes` and, if any are notable, generate a `div` with all the bird names from notable records. See Section 23.8, "`MonthCell.__notablesBlock()`: Display any notable records" (p. 56).

noteweb

```
#-- 5 --
# [ if dayNotes contains any notable records ->
#       li += a div element containing the bird names
#       from notable records
#       else -> I ]
self.__notablesBlock ( li, dayNotes )
```

## 23.8. MonthCell.\_\_notablesBlock(): Display any notable records

This method looks through the records inside a `birdnotes.DayNotes` instance to see if any of the records are flagged as notable. If so, it generates a `div` element containing a list of their names.

noteweb

```
# - - - M o n t h C e l l . _ _ n o t a b l e s B l o c k

def __notablesBlock ( self, parent, dayNotes ):
    '''If any records are notable, display their names.

    [ (parent is an et.Element) and
      (dayNotes is a DayNotes instance) ->
        if any records within dayNotes are flagged as notable ->
          parent += a div element containing the names
                  from all notable records
        else -> I ]
    ...
    '''
```

First we'll go through the `BirdForm` instances inside `dayNotes` and accumulate a list of those flagged as notable. If the list is empty, we're done.

noteweb

```
#-- 1 --
notablesList = [ birdForm
                 for birdForm in dayNotes.genForms()
                 if birdForm.notable ]

#-- 2 --
if len(notablesList) == 0:
    return
```

The block is a “`div class='loc-child'`” element. Its first child is a `span class='notable'` containing the label “Notable:”. The bird names follow, separated by commas.

noteweb

```
#-- 3 --
# [ parent += a new div element with class=LOC_CHILD_CLASS
#   div := that new element ]
div = et.SubElement ( parent, 'div' )
div.attrib['class'] = LOC_CHILD_CLASS
```

See Section 23.9, “`MonthCell.__span(): Add a span inline`” (p. 57).

noteweb

```
#-- 4 --
# [ div += a new span element with class=NOTABLE_CLASS,
#   containing the text 'Notable: ', plus a space ]
self.__span ( div, NOTABLE_CLASS, 'Notable: ' )
tccpage2.addTextMixed ( div, NBSP )

#-- 5 --
# [ div += names of the birds in notablesList,
#   separated by ', ' ]
nameList = [ str(birdForm.birdId)
             for birdForm in notablesList ]
tccpage2.addTextMixed ( div, '; '.join ( nameList ) )
```

## 23.9. MonthCell.\_\_span(): Add a span inline

This service routine is used to add to some parent element a span child with a given class and textual content.

noteweb

```
# - - -   M o n t h C e l l . _ _ s p a n

def __span ( self, parent, class_, text ):
    '''Add text inside a span inline

        [ (parent is an et.Element) and
          (class_ is a string) and
          (text is a string) ->
            parent += a new span element with class=(class_)
                      and text=(text)
            return that new span element ]
    ...
    span = et.SubElement ( parent, 'span' )
    span.attrib['class'] = class_
    span.text = text
    return span
```

## 23.10. MonthCell.\_\_dayBlock(): Render one daily note set

Each DayNotes instance is rendered into these XHTML elements:

- A horizontal rule, hr.
- An h2 title that defines the given anchor. See Section 23.11, “MonthCell.\_\_dayTitle(): Render the title for one daily block” (p. 58).
- A daily summary block. See Section 23.12, “MonthCell.\_\_daySummary(): Render the daily summary block” (p. 58).
- Rendering of the BirdForm children. See Section 23.19, “MonthCell.\_\_birdForm(): Render one BirdForm” (p. 64).

noteweb

```
# - - -   M o n t h C e l l . _ _ d a y B l o c k

def __dayBlock ( self, parent, dayNotes, anchor ):
    '''Generate all the output for one DayNotes instance.
    ...
    #-- 1 --
    # [ parent += an empty hr element ]
    et.SubElement ( parent, 'hr' )

    #-- 2 --
    # [ parent += an h2 heading describing dayNotes
    #       that has id=anchor ]
    self.__dayTitle ( parent, dayNotes, anchor )

    #-- 3 --
    # [ parent += a div class=DAY_SUMMARY_CLASS element
    #       representing the daily summary for dayNotes ]
```

```

self.__daySummary ( parent, dayNotes )

#-- 4 --
# [ parent += XHTML for all the BirdForm children
#       of dayNotes ]
for birdForm in dayNotes.genForms():
    #-- 4 body --
    # [ parent += XHTML rendering of birdForm ]
    self.__birdForm ( parent, birdForm )

```

## 23.11. MonthCell.\_\_dayTitle(): Render the title for one daily block

noteweb

```

# - - - M o n t h C e l l . _ _ d a y T i t l e

def __dayTitle ( self, parent, dayNotes, anchor ):
    '''Render the h2 title and define the anchor.

    [ (parent is an et.Element) and
      (dayNotes is a DayNotes instance) and
      (anchor is a valid HTML anchor string) ->
        parent += an h2 heading describing dayNotes
                  that has id=anchor ]
    ...

#-- 1 --
h2 = et.SubElement ( parent, 'h2', id=anchor )
h2.text = dayNotes.title()

```

## 23.12. MonthCell.\_\_daySummary(): Render the daily summary block

This method renders the day-summary content for one day-notes.

noteweb

```

# - - - M o n t h C e l l . _ _ d a y S u m m a r y

def __daySummary ( self, parent, dayNotes ):
    '''Render the daily summary block.

    [ (parent is an et.Element) and
      (dayNotes is a DayNotes instance) ->
        parent += a div class=DAY_SUMMARY_CLASS element
                  representing the daily summary for dayNotes ]
    ...

```

The entire content of this block is wrapped inside `div class='day-summary'`. If there are any notable records for the day, they are added just inside it: see Section 23.8, “MonthCell.\_\_notablesBlock(): Display any notable records” (p. 56).

noteweb

```

#-- 1 --
# [ parent := parent with a new div element added with
#         class=DAY_SUMMARY_CLASS
#         summaryDiv := that div element ]

```

```

summaryDiv = et.SubElement ( parent, 'div' )
summaryDiv.attrib['class'] = DAY_SUMMARY_CLASS

#-- 2 --
# [ if dayNotes contains any notable records ->
#     summaryDiv += a div element containing the
#                 names from all notable records
# else -> I ]
self.__notablesBlock ( summaryDiv, dayNotes )

```

Next we list the localities for the day; see Section 23.13, “MonthCell.\_\_locDef(): Display a locality definition” (p. 59).

noteweb

```

#-- 3 --
# [ summaryDiv += blocks displaying the localities
#     in dayNotes.daySummary ]
for loc in dayNotes.daySummary.genLocs():
    self.__locDef ( summaryDiv, loc )

```

Rounding out the daily summary block are the day-annotation elements; see Section 23.14, “MonthCell.\_\_dayAnnotation(): Render day-annotation content” (p. 61).

noteweb

```

#-- 4 --
# [ summaryDiv += blocks displaying day-annotation
#     content from dayNotes.daySummary, if any ]
self.__dayAnnotation ( summaryDiv, dayNotes.daySummary )

```

### 23.12.1. Historical note

In the output of the XSLT script that was the predecessor of *noteweb*, it was tedious to determine the location code for any given bird form:

- The default location code was displayed at the top of the daily summary block, followed by the list of that day’s location codes and their definitions.
- Under each bird form, the location code was displayed only for records that were not at the default location.

So the reader’s process for determining the locality of a record was:

- If the bird form record shows no location, find the location code shown as the “Default location,” then look up the code in the table of locations.
- If the bird form record shows a location code, look that up in the table of locations.

A better solution is to show the full-length locality name on each bird form record. This means we don’t need to show the default location code at all; it’s just an artifact of the file encoding (for the convention of the data entry worker).

The only reason for the old, ugly rendering was that the author found it difficult to implement in straight XSLT. Have a look at the old `birdhtml.xsl` if you like. It was put up as a stopgap before the `bird-notes.py` package was written.

## 23.13. MonthCell.\_\_locDef(): Display a locality definition

See Section 4.5, “XHTML for locality definitions” (p. 8).

```
# - - -   M o n t h C e l l . _ _ l o c D e f

def __locDef ( self, parent, loc ):
    '''Display the definition of one locality.

    [ (parent is an et.Element) and
      (loc is a birdnotes.Loc instance) ->
      parent += a div element displaying loc ]
    ...
```

All the content generated by this method is wrapped in a `div class=LOC_DEF_CLASS`. In each case, the first content is “@” followed by the locality's code, a colon, and the locality's name.

```
#-- 1 --
# [ parent += a div element with class=LOC_DEF_CLASS
#   topDiv := that div element ]
topDiv = et.SubElement ( parent, 'div' )
topDiv.attrib['class'] = LOC_DEF_CLASS

#-- 2 --
# [ topDiv += '@' + loc.code + ': ' + loc.name ]
topDiv.text = '@%s: %s' % (loc.code, loc.name )
```

If there is a narrative block for this locality, render it next inside a `div class=LOC_NARRATIVE_CLASS`. The text is just a string, not an instance of the `Narrative` class.

```
#-- 3 --
# [ if bool(loc.text) ->
#   topDiv += loc.text, wrapped in a div element
#         with class=LOC_NARRATIVE_CLASS
#   else -> I ]
if loc.text:
    narraDiv = et.SubElement ( topDiv, 'div' )
    narraDiv.attrib['class'] = LOC_NARRATIVE_CLASS
    narraDiv.text = loc.text
```

The list of GPS waypoints follows, if there are any.

```
#-- 4 --
# [ topDiv += a sequence of divs with class
#   LOC_NARRATIVE_CLASS containing GPS descriptions
#   from loc, if any ]
for gps in loc.genGps():
    narraDiv = et.SubElement ( topDiv, 'div' )
    narraDiv.attrib['class'] = LOC_NARRATIVE_CLASS
    narraDiv.text = ( 'GPS: %s %s' %
                     (gps.waypoint, gps.text) )
```

## 23.14. MonthCell.\_\_dayAnnotation(): Render day-annotation content

noteweb

```
# - - - M o n t h C e l l . _ _ d a y A n n o t a t i o n

def __dayAnnotation ( self, parent, daySummary ):
    '''Render day-annotation content.

    [ (parent is an et.Element) and
      (daySummary is a birdnotes.DaySummary instance) ->
        parent += blocks displaying day-annotation
        content from daySummary, if any ]
    ...
    '''
```

This method generates the various types of labeled annotation first in a standard order—route, weather, missed, and film—followed by unclassified notes. For the general structure, see Section 4.6, “XHTML for day-annotation elements” (p. 9). See also Section 23.15, “MonthCell.\_\_annoBlock(): Annotation block with a label” (p. 61).

noteweb

```
#-- 1 --
# [ if daySummary.route is not None ->
#   parent += a div labeled with 'route' displaying
#           daySummary.route
#   else -> I ]
if daySummary.route is not None:
    self.__annoBlock ( parent, 'Route', daySummary.route )

#-- 2 --
if daySummary.weather is not None:
    self.__annoBlock ( parent, 'Weather', daySummary.weather )

#-- 3 --
if daySummary.missed is not None:
    self.__annoBlock ( parent, 'Missed', daySummary.missed )

#-- 4 --
if daySummary.film is not None:
    self.__annoBlock ( parent, 'Film', daySummary.film )
```

The handling of the unlabeled narrative is slightly different: it is not enclosed in a block with a label, so it goes straight to Section 23.16, “MonthCell.\_\_narrative(): Render a Narrative instance” (p. 62).

noteweb

```
#-- 5 --
if daySummary.notes is not None:
    self.__narrative ( parent, daySummary.notes )
```

## 23.15. MonthCell.\_\_annoBlock(): Annotation block with a label

This method generates one of the blocks displaying specific kinds of day-annotation data such as “Route”. For the generated output, see Section 4.6, “XHTML for day-annotation elements” (p. 9).

```
# - - -   M o n t h C e l l . _ _ a n n o B l o c k

def __annoBlock ( self, parent, label, narr ):
    '''Generate a label block of day annotation.

        [ (parent is an et.Element) and
          (label is a string) and
          (narr is a Narrative instance) ->
            parent += a div with class=LOC_NARRATIVE_CLASS
                    containing label and an XHTML rendering of narr ]
        ...
    '''
    #-- 1 --
    # [ parent += a div element with class=LOC_NARRATIVE_CLASS
    #   div := that div element ]
    div = et.SubElement ( parent, 'div' )
    div.attrib['class'] = LOC_NARRATIVE_CLASS
```

The block's label is wrapped in a `span`; see Section 5.2, “Inline markup rules” (p. 12) and Section 23.9, “`MonthCell.__span()`: Add a span inline” (p. 57).

```
#-- 2 --
# [ div += a span element with class=LOC_LABEL_CLASS
#   containing label ]
self.__span ( div, LOC_LABEL_CLASS, "%s " % label )
```

The content of the `div` depends on whether the given `narr` has one paragraph or multiple paragraphs.

- If `narr` has only one child paragraph, we go directly to Section 23.18, “`MonthCell.__paraContent()`: Content of one paragraph” (p. 63) to avoid getting another level of `div` elements around the content.
- In the multi-paragraph case, separate child `div` elements are generated for each paragraph by Section 23.16, “`MonthCell.__narrative()`: Render a Narrative instance” (p. 62).

```
#-- 3 --
# [ div += XHTML rendering of narr ]
if len(narr) == 1:
    #-- 3.1 --
    # [ div += contents of the first paragraph of narr ]
    self.__paraContent ( div, narr[0] )
else:
    #-- 3.2 --
    # [ div += child div elements containing the
    #   paragraphs of narr ]
    self.__narrative ( div, narr )
```

## 23.16. `MonthCell.__narrative()`: Render a Narrative instance

The rendering of a `Narrative` instance is straightforward: see Section 4.7, “XHTML rendering of narrative elements” (p. 9). Each contained `Paragraph` is rendered in a `div class=PARA_CLASS`.

```
# - - -   M o n t h C e l l . _ _ n a r r a t i v e

def __narrative ( self, parent, narr ):
    '''Render a birdnotes.Narrative instance into XHTML.

    [ (parent is an et.Element) and
      (narr is a birdnotes.Narrative instance) ->
        parent += XHTML rendering of narr ]
    ...
    #-- 1 --
    for para in narr.genParas():
        #-- 1 body --
        # [ parent += a div class=PARA_CLASS element
          #   containing the XHTML rendering of para ]
        self.__paragraph ( parent, para )
```

### 23.17. MonthCell.\_\_paragraph(): Render one paragraph

This method renders a `birdnotes.Paragraph` instance as an ordinary text paragraph.

```
# - - -   M o n t h C e l l . _ _ p a r a g r a p h

def __paragraph ( self, parent, para ):
    '''Add one paragraph of narrative.

    [ (parent is an et.Element) and
      (para is a birdnotes.Paragraph instance) ->
        parent += a div class=PARA_CLASS containing
          the XHTML rendering of para ]
    ...
```

Everything is wrapped in a `div class=PARA_CLASS`.

```
#-- 1 --
# [ parent += a div element with class=PARA_CLASS
#   div := that div element ]
div = et.SubElement ( parent, 'div' )
div.attrib['class'] = PARA_CLASS
```

For the translation of the paragraph's content, see Section 23.18, “`MonthCell.__paraContent()`: Content of one paragraph” (p. 63).

```
#-- 2 --
# [ div += div elements containing XHTML rendering
#   of the content of para ]
self.__paraContent ( div, para )
```

### 23.18. MonthCell.\_\_paraContent(): Content of one paragraph

This method translates the “phrase list” structure inside a `Paragraph` instance into XHTML.

Although some text in a paragraph can be marked up (e.g., with `genus` tags), the structure is only one level deep: the child elements do not themselves have element children.

However, the rendering of even this shallow structure is not completely obvious, due to the strange way that the `lxml` package handles mixed content: text after an element is stored in the `.tail` attribute of the preceding element, and not associated with the parent (as it would be in the Document Object Model).

Fortunately, the `tccpage2` module already contains a function named `addTextMixed()` that handles this problem. For details of this function, see the documentation for `tccpage2`<sup>15</sup>.

For a `Paragraph` element `P`, method `P.genContent()` produces a list of `(tag, text)` tuples.

- If the `tag` is `None`, the `text` is plain text, not marked up. We use `tccpage2.addTextMixed()` to add it to the structure wherever the next text string goes.
- If the `tag` is not `None`, its value becomes the `class` attribute of a `span` element containing the `text`.

noteweb

```
# - - -   M o n t h C e l l . _ _ p a r a C o n t e n t

def __paraContent ( self, parent, para ):
    """Translate the content inside one paragraph.

    [ (parent is an et.Element) and
      (para is a birdnotes.Paragraph instance) ->
      parent += XHTML rendering of para's contents ]
    """
    #-- 1 --
    for tag, s in para.genContent():
        #-- 1 body --
        # [ (tag is None or a tag string) and
          # (s is a string) ->
          #   if tag is None ->
          #     parent := parent with (s) added to its text
          #   else ->
          #     parent := parent with a new span element
          #                 added, with class=(tag), and text (s) ]
        if tag is None:
            tccpage2.addTextMixed ( parent, s )
        else:
            self.__span ( parent, tag, s )
```

## 23.19. `MonthCell.__birdForm()`: Render one `BirdForm`

For an overview of the generated output, see Section 4.8, “XHTML for the form element” (p. 9). A `div class=FORM_CLASS` wraps all generated content, except for notable records, which get `class=NOTABLE_FORM_CLASS`.

noteweb

```
# - - -   M o n t h C e l l . _ _ b i r d F o r m

def __birdForm ( self, parent, birdForm ):
    """Output all the data from one BirdForm instance.
```

<sup>15</sup> <http://infohost.nmt.edu/tcc/projects/tccpage2/addTextMixed.html>

```

    [ (parent is an et.Element) and
      (birdForm is a birdnotes.BirdForm instance) ->
      parent += XHTML rendering of birdForm ]
    ...
#-- 1 --
# [ parent += a new div element with class=FORM_CLASS
#   div := that div element ]
div = et.SubElement ( parent, 'div' )
if birdForm.notable:
    div.attrib['class'] = NOTABLE_FORM_CLASS
else:
    div.attrib['class'] = FORM_CLASS

```

Next comes the name of this bird form. We'll wrap it in a `span class='notable'` if the `birdForm.notable` flag is true, otherwise we'll use the regular `span class='bird-name'`.

The name of the bird form is a `birdnotes.BirdId` instance in `birdForm.birdId`; applying the `str()` function to a `BirdId` gives the full name, even for compound forms such as 'Gadwall x Mallard'. See Section 23.9, "MonthCell.\_\_span(): Add a span inline" (p. 57).

noteweb

```

#-- 2 --
# [ if birdForm.notable ->
#   div += birdForm.birdId's name, wrapped in a
#         span class=NOTABLE_CLASS
#   else ->
#   div += birdForm.birdId's name, wrapped in a
#         span class=BIRD_NAME_CLASS ]
if birdForm.notable:
    class_ = NOTABLE_CLASS
else:
    class_ = BIRD_NAME_CLASS
self.__span ( div, class_, birdForm.birdId.engComma() )
tccpage2.addTextMixed ( div, NBSP )

```

At this point, the rendering depends on whether this one sighting or multiple sightings:

- If a single sighting, we want the `age-sex-group` content on the same line as the bird name. If there is `loc-group` or `sighting-notes` content, it should follow in separate `div` elements. See Section 23.20, "MonthCell.\_\_singleSighting(): Single-sighting case" (p. 66).
- For multiple sightings, there may be `loc-group` or `sighting-notes` content attached both at the form level and at the `floc` level.

In that case, we must render any `loc-group` or `sighting-notes` content at the form level first. Then, each `floc` element is rendered in the order `age-sex-group`, `loc-detail`, and `sighting-notes`. See Section 23.21, "MonthCell.\_\_multiSighting(): Multiple-sighting case" (p. 67).

noteweb

```

#-- 3 --
# [ if birdForm contains one sighting ->
#   div += (that sighting's age-sex-group) +
#         (that sighting's loc-group, if any) +
#         (that sighting's sighting-notes, if any)
#   else ->
#   div += (birdForm's loc-group content, if any) +

```

```

#         (birdForm's sighting-notes, if any) +
#         (birdForm's sightings packaged in separate divs) ]
if len(birdForm) == 1:
    self.__singleSighting ( div, birdForm )
else:
    self.__multiSighting ( div, birdForm )

```

## 23.20. MonthCell.\_\_singleSighting(): Single-sighting case

See the notes about overall structure in Section 23.19, “MonthCell.\_\_birdForm(): Render one BirdForm” (p. 64).

noteweb

```

# - - -   M o n t h C e l l . _ _ s i n g l e S i g h t i n g

def __singleSighting ( self, parent, birdForm ):
    '''Render the single-sighting case of a bird form.

    [ (parent is an et.element) and
      (birdForm is a birdnotes.BirdForm instance with
       one sighting) ->
        parent += (that sighting's age-sex-group) +
                  (that sighting's loc-group, if any) +
                  (that sighting's sighting-notes, if any) ]
    ...

```

See Section 23.26, “MonthCell.\_\_ageSexGroup(): Render age-sex-group content” (p. 71).

noteweb

```

#-- 1 --
# [ sight := the first child sighting of birdForm ]
sight = birdForm[0]

#-- 2 --
# [ if sight.ageSexGroup is not None ->
#   parent += XHTML rendering of sight.ageSexGroup
#   else -> I ]
if sight.ageSexGroup is not None:
    self.__ageSexGroup ( parent, sight.ageSexGroup )

```

See Section 23.22, “MonthCell.\_\_locGroup(): Render a locality group” (p. 67).

noteweb

```

#-- 3 --
# [ parentn += XHTML rendering of sight's effective locality ]
self.__locGroup ( parent, sight.getLocGroup() )

```

See Section 23.23, “MonthCell.\_\_sightNotes(): Render a sighting notes group” (p. 68).

noteweb

```

#-- 4 --
# [ if sight.sightNotes is not None ->
#   parent += XHTML rendering of sight.sightNotes
#   else -> I ]
if sight.sightNotes is not None:
    self.__sightNotes ( parent, sight.sightNotes )

```

## 23.21. MonthCell.\_\_multiSighting(): Multiple-sighting case

See the general remarks in Section 23.19, “MonthCell.\_\_birdForm(): Render one BirdForm” (p. 64).

noteweb

```
# - - - M o n t h C e l l . _ _ m u l t i S i g h t i n g

def __multiSighting ( self, div, birdForm ):
    '''Render the multi-sighting case of a bird form.

    [ (div is an et.Element) and
      (birdForm is a birdnotes.birdForm instance) ->
        div += (birdForm's loc-group content, if any) +
                (birdForm's sighting-notes, if any) +
                (birdForm's sightings packaged in separate divs) ]
    ...
    '''
```

See Section 23.22, “MonthCell.\_\_locGroup(): Render a locality group” (p. 67). We want to display this only if there is locality data explicitly attached at this level; if the locality is all inherited, the children will take care of displaying their localities.

noteweb

```
#-- 1 --
# [ if birdForm.locGroup is not None ->
#   div += XHTML rendering of birdForm.locGroup
#   else -> I ]
if birdForm.locGroup is not None:
    self.__locGroup ( div, birdForm.locGroup )
```

See Section 23.23, “MonthCell.\_\_sightNotes(): Render a sighting notes group” (p. 68).

noteweb

```
#-- 2 --
# [ if birdForm.sightNotes is not None ->
#   div += XHTML rendering of birdForm.sightNotes ]
if birdForm.sightNotes:
    self.__sightNotes ( div, birdForm.sightNotes )
```

Everything else we need to generate comes from child sightings of birdForm. See Section 23.25, “MonthCell.\_\_floc(): Generate one of multiple sightings” (p. 70).

noteweb

```
#-- 3 --
# [ div += birdForm's sightings packaged in separate
#   divs ]
for sight in birdForm.genSightings():
    #-- 3 body --
    # [ div += sight packaged in a separate div ]
    self.__floc ( div, sight )
```

## 23.22. MonthCell.\_\_locGroup(): Render a locality group

For a discussion of how this method generates both inline and block content, see Section 4.10, “XHTML rendering of form data” (p. 10).

```
# - - -   M o n t h C e l l . _ l o c G r o u p

def __locGroup ( self, parent, locGroup ):
    '''Render a LocGroup instance in XHTML.

    [ (parent is an et.Element) and
      (locGroup is a birdnotes.LocGroup instance) ->
      parent += XHTML rendering of locGroup ]
    ...
```

The `tccpage2.addTextMixed()` places the location name as inline text. For documentation for `tccpage2`, see Section 3, “Overview of the internals” (p. 4).

```
#-- 1 --
# [ if locGroup.loc is not None ->
#   parent += locGroup.loc.name
#   else -> I ]
if locGroup.loc is not None:
    tccpage2.addTextMixed ( parent, '@%s' % locGroup.loc.name )

#-- 2 --
# [ if locGroup.gps is not None ->
#   parent += (' GPS: ')+(locGroup.gps)
#   else -> I ]
if locGroup.gps is not None:
    tccpage2.addTextMixed ( parent,
        ' GPS: %s' % locGroup.gps )
```

If there is `loc-detail` content, it is a `Narrative` instance; see Section 23.16, “`MonthCell.__narrative(): Render a Narrative instance`” (p. 62).

```
#-- 3 --
# [ if locGroup.locDetail is not None ->
#   parent += locGroup.locDetail wrapped in a div
#   else -> I ]
if locGroup.locDetail is not None:
    self.__narrative ( parent, locGroup.locDetail )
```

## 23.23. `MonthCell.__sightNotes(): Render a sighting notes group`

For an overview of the generated output, see Section 4.11, “XHTML rendering of sighting notes” (p. 11).

```
# - - -   M o n t h C e l l . _ _ s i g h t N o t e s

def __sightNotes ( self, parent, sightNotes ):
    '''Render sighting-notes content into XHTML.

    [ (parent is an et.Element) and
      (sightNotes is a birdnotes.SightNotes instance) ->
      parent += XHTML rendering of birdForm.sightNotes ]
    ...
```

We'll use the method described in Section 23.15, “MonthCell.\_\_annoBlock(): Annotation block with a label” (p. 61) to generate div elements with run-in initial labels.

noteweb

```
#-- 1 --
# [ if sightNotes.desc is not None ->
#   parent += XHTML rendering of sightNotes.desc
#   else -> I ]
if sightNotes.desc is not None:
    self.__annoBlock ( parent, 'Description:', sightNotes.desc )

#-- 2 --
# [ simile ]
if sightNotes.behavior is not None:
    self.__annoBlock ( parent, 'Behavior:', sightNotes.behavior )

#-- 3 --
if sightNotes.voc is not None:
    self.__annoBlock ( parent, 'Vocalizations:',
                      sightNotes.voc )

#-- 4 --
if sightNotes.breeding is not None:
    self.__annoBlock ( parent, 'Breeding:',
                      sightNotes.breeding )
```

Next come the photos. If there are any, they are placed in a child div class='para' below.

noteweb

```
#-- 5 --
# [ parent += rendering of photos in sightNotes, if any ]
photoList = [ x for x in sightNotes.genPhotos() ]
if len(photoList) > 0:
    photoDiv = et.SubElement ( parent, 'div' )
    photoDiv.attrib['class'] = PARA_CLASS
    for photo in photoList:
        self.__photo ( photoDiv, photo )
```

The sightNotes.notes content, if any, is not labeled; see Section 23.16, “MonthCell.\_\_narrative(): Render a Narrative instance” (p. 62).

noteweb

```
#-- 6 --
if sightNotes.notes is not None:
    self.__narrative ( parent, sightNotes.notes )
```

## 23.24. MonthCell.\_\_photo(): Generate a photo reference or link

For the XHTML generated here, see Section 4.12, “XHTML rendering of photo links” (p. 12).

noteweb

```
# - - -   M o n t h C e l l . _ _ p h o t o

def __photo ( self, parent, photo ):
    '''Render one photo instance: a link, or just the catalog number.
```

```

    [ (parent is an et.Element) and
      (photo is a birdnotes.Photo instance) ->
        parent += XHTML rendering of photo ]
    ...

```

If there is an URL associated with this photo, generate a link to it, otherwise just spit out the catalog number unadorned.

noteweb

```

#-- 1 --
# [ if photo.url is None ->
#   parent += photo.catNo
#   return
# else -> I ]
if photo.url is None:
    tccpage2.addTextMixed ( parent, "%s " % photo.url )
    return

#-- 2 --
# [ parent += a new 'a' element with href=photo.url
#   a := that new element ]
a = et.SubElement ( parent, 'a', href=photo.url )

#-- 3 --
# [a += a new 'img' element with src=(thumbnail for
#   photo.catNo) and alt=photo.catNo ]
thumbUrl = '/~shipman/thumb/%s.jpg' % photo.catNo
img = et.SubElement ( a, 'img', alt=photo.catNo,
                     src=thumbUrl )

```

## 23.25. MonthCell.\_\_floc(): Generate one of multiple sightings

When there are multiple sightings of a form, each is packaged in its own div element with class=FLOC\_CLASS. For general notes on rendering, see Section 23.19, “MonthCell.\_\_birdForm(): Render one BirdForm” (p. 64).

noteweb

```

# - - - M o n t h C e l l . _ _ f l o c

def __floc ( self, parent, sight ):
    '''Render one sighting in the multi-sighting case.

    [ (parent is an et.Element) and
      (sight is a birdnotes.sighting instance) ->
        parent += XHTML rendering of sight, packaged in
          a separate div ]
    ...

#-- 1 --
# [ parent += a new div element with class=FLOC_CLASS
#   div := that div element ]
div = et.SubElement ( parent, 'div' )
div.attrib['class'] = FLOC_CLASS

```

See Section 23.26, “MonthCell.\_\_ageSexGroup(): Render age-sex-group content” (p. 71).

```

#-- 2 --
# [ if sight.ageSexGroup is not None ->
#   div += XHTML rendering of sight.ageSexGroup
#   else -> I ]
if sight.ageSexGroup is not None:
    self.__ageSexGroup ( div, sight.ageSexGroup )

```

See Section 23.22, “MonthCell.\_\_locGroup(): Render a locality group” (p. 67).

```

#-- 3 --
# [ div += XHTML rendering of sight's effective locality ]
self.__locGroup ( div, sight.getLocGroup() )

```

See Section 23.23, “MonthCell.\_\_sightNotes(): Render a sighting notes group” (p. 68).

```

#-- 4 --
# [ if sight.sightNotes is not None ->
#   div += XHTML rendering of sight.sightNotes
#   else -> I ]
if sight.sightNotes is not None:
    self.__sightNotes ( div, sight.sightNotes )

```

## 23.26. MonthCell.\_\_ageSexGroup(): Render age-sex-group content

For general notes on the rendering of a `birdnotes.AgeSexGroup` instance, see Section 4.10, “XHTML rendering of form data” (p. 10).

```

# - - -   M o n t h C e l l . _ _ a g e S e x G r o u p

SEX_CODE_MAP = { 'u': u'', 'm': u'\u2642', 'f': u'\u2640' }

def __ageSexGroup ( self, parent, ageSexGroup ):
    """Render a birdnotes.AgeSexGroup instance into XHTML.

    [ (parent is an et.Element) and
      (ageSexGroup is a birdNotes.AgeSexGroup instance) ->
      parent += XHTML rendering of sight.ageSexGroup ]
    """

```

For general notes on the generated output, see Section 4.10, “XHTML rendering of form data” (p. 10). First we render the count of individuals, if any.

```

#-- 1 --
# [ if ageSexGroup.count is not None ->
#   parent += ageSexGroup.count
#   else -> I ]
if ageSexGroup.count is not None:
    tccpage2.addTextMixed ( parent, ageSexGroup.count )

```

The age code comes next. Code 'p' is rendered as Greek letter phi ( $\phi$ ).

```

#-- 2 --
# [ if ageSexGroup.age is None ->
#   I
#   else if ageSexGroup.age is 'p' ->
#     parent += Greek letter phi
#   else ->
#     parent += ageSexGroup.age ]
if ageSexGroup.age is not None:
    if ageSexGroup.age == 'p':
        tccpage2.addTextMixed ( parent, PHI )
    else:
        tccpage2.addTextMixed ( parent, ageSexGroup.age )

```

Add the sex code, questionability status, and *fide* details. Although the Unicode entities for the male symbol "" (&#x2642;) and female symbol "" (&#x2640;) don't currently render correctly in the PDF for this document, as of May 2011 all the browsers I checked (Camino, Chrome, Firefox, Opera, and Safari) do correctly display these symbols.

```

#-- 3 --
# [ if ageSexGroup.sex is not None ->
#   parent += ageSexGroup.sex
#   else -> I ]
if ageSexGroup.sex is not None:
    tccpage2.addTextMixed ( parent,
        self.SEX_CODE_MAP[ageSexGroup.sex] )

#-- 4 --
# [ if ageSexGroup.q is '?' ->
#   parent += '?'
#   else if ageSexGroup.q is '-' ->
#     parent += ' [uncountable]'
#   else -> I ]
if ageSexGroup.q is '?':
    tccpage2.addTextMixed ( parent, '?' )
elif ageSexGroup.q is '-':
    tccpage2.addTextMixed ( parent, ' [uncountable]' )

#-- 5 --
# [ if ageSexGroup.fide is not None ->
#   parent += '['+(a span element of class=GENUS_CLASS
#     containing 'fide')+'+ageSexGroup.fide+']'
if ageSexGroup.fide is not None:
    tccpage2.addTextMixed ( parent, '[' )
    self.__span ( parent, GENUS_CLASS, 'fide' )
    tccpage2.addTextMixed ( parent, '%s]' % ageSexGroup.fide )

#-- 6 --
# [ parent += ' ' ]
tccpage2.addTextMixed ( parent, ' ' )

```

## 23.27. MonthCell.monthName(): Translate a month key to a month name (static method)

This static method takes a month key of the form 'yyyy-mm' and returns the corresponding text 'monthName, yyyy'. For the mapping from month numbers to month names, see Section 10.1, "MONTH\_NAME\_MAP: Translate month numbers to month names" (p. 19).

noteweb

```
# @staticmethod
def monthName ( yyyy_mm ):
    '''Translate a month key to a month name.

    [ yyyy_mm is a month key as 'yyyy-mm' ->
      return the month name as 'monthName yyyy' ]
    ...
    #-- 1 --
    # [ yyyy := the year part
    #   mm   := the month part ]
    #--
    # 01234567
    # yyyy-mm
    #--
    yyyy = yyyy_mm[:4]
    mm   = yyyy_mm[5:7]

    #-- 2 --
    return '%s %s' % (MONTH_NAME_MAP[mm], yyyy)

monthName = staticmethod ( monthName )
```

## 24. Epilogue

noteweb

```
#####
# Epilogue
#-----
if __name__ == '__main__':
    main()
```

## 25. Defects discovered

This is a list of the defects discovered since first execution of the script. They are divided into three categories:

- Section 25.1, "Syntax errors" (p. 74): Ordinary syntax errors.
- Section 25.2, "Logic errors" (p. 74): Logic errors.
- Section 25.3, "Run-time type matching errors" (p. 76): Errors that would have been detected at compile time in languages with strong compile-time type checking, but which emerged at execution time due to Python's run-time type system.

## 25.1. Syntax errors

1. In `buildIndex()`, neglected to close parentheses:

```
print >>sys.stderr, ( '*** Can't write the index page '
    '%s'." % indexName
```

should be:

```
print >>sys.stderr, ( "*** Can't write the index page "
    '%s'." % indexName )
```

2. In `indexBoilerplate`, an unclosed string constant:

```
conLink.tail = ( " for notational conventions and the "
    "author's contact information. )
```

should be:

```
conLink.tail = ( " for notational conventions and the "
    "author's contact information." )
```

3. In `YearRow.readOneMonth()`, unclosed parenthesis.

```
print >>sys.stderr, ( "*** Invalid monthly file "
    "'%s': %s" % (monthFileName, detail) )
```

should be:

```
print >>sys.stderr, ( "*** Invalid monthly file "
    "'%s': %s" % (monthFileName, detail) )
```

4. In `MonthCell.__notablesBlock()`, unclosed string constant.

```
self.__span ( div, NOTABLE_CLASS, 'Notable: )
```

should be:

```
self.__span ( div, NOTABLE_CLASS, 'Notable:' )
```

## 25.2. Logic errors

1. In `YearRow.readOneMonth()`, neglected to prepend the year number and a slash to the actual file name passed to `birdNoteSet.readFile()`.
2. There's no need to add an `h1` heading in the page body, because `tccpage2` supplies one already. Remove this logic from `indexBoilerPlate()`.
3. In `YearRow.readOneMonth()`, the intended function was correct, but the code was wrong:

```
# [ self.__monthMap[mm] := a new MonthCell instance
#     with self as the parent, month=mm, and
#     birdNoteSet=birdNoteSet ]
monthCell = MonthCell ( self, mm, birdNoteSet )
```

should be:

```
self.__monthMap[mm] = MonthCell ( self, mm, birdNoteSet )
```

4. In `buildRow()`, neglected to place the year into the cell.

```
# [ tr += a th element with class=ROW_LABEL_CLASS
#         containing yearRow.yyyy ]
rowLabel = et.SubElement ( tr, 'th' )
rowLabel.attrib['class'] = ROW_LABEL_CLASS
```

Add this line:

```
rowLabel.text = yearRow.yyyy
```

5. In `YearCollection.findNext()`, this logic returned a month number 'mm' when it should have returned a full key 'yyyy-mm'.

```
# [ if any year with a key in yyyyList[pos+1:] has at
#   least one month in it ->
#   return the yyyy-mm key of the first month in
#   the first such year
#   else -> I ]
for nextYear in yyyyList[pos+1:]:
    nextRow = self[nextYear]
    if len(nextRow) > 0:
        return nextRow.firstMonth()
```

Make that last line:

```
firstMM = nextRow.firstMonth()
return '%s-%s' % (nextYear, firstMM)
```

Symmetrical fix in `YearCollection.findPrev()`:

```
if len(prevRow) > 0:
    lastMM = prevRow.lastMonth()
    return '%s-%s' % (prevYear, lastMM)
```

6. `MonthCell.__pageFrame()` completely neglected to consider the case where either the previous or next page link is `None`. Pretty much rewrote the whole method.
7. In `MonthCell.__sightNotes()`, a cut-and-paste error:

```
if sightNotes.voc is not None:
    self.__annoBlock ( parent, 'Vocalizations:',
                       sightNotes.behavior )
```

The last statement should be:

```
self.__annoBlock ( parent, 'Vocalizations:',
                  sightNotes.voc )
```

Several quite similar errors in `MonthCell.__dayAnnotation`:

```
#-- 2 --
if daySummary.weather is not None:
    self.__annoBlock ( parent, 'Weather', daySummary.route )
```

```
#-- 3 --
if daySummary.missed is not None:
    self.__annoBlock ( parent, 'Missed', daySummary.route )
```

8. Completely forgot about photo links.

## 25.3. Run-time type matching errors

1. In `YearCollection.addYear()`, the call to the `YearRow` constructor is missing the second (`txny`) argument, because it was written before the constructor was fleshed out.

```
yearRow = YearRow ( self, yyyy )
```

should be:

```
yearRow = YearRow ( self, txny, yyyy )
```

which means of course that the `YearCollection.addYear()` method also needs this argument, and the call to that method in `findYears()` must also supply it.

2. In `YearRow.readOneMonth()`, the reference to the `BirdNoteSet` constructor was not properly qualified with its containing module name.

```
# [
    birdNoteSet := a new BirdNoteSet instance with #
    taxonomy self.txny ] birdNoteSet = BirdNoteSet ( txny
)
```

should be:

```
# [ birdNoteSet := a new birdnotes.BirdNoteSet instance
#     with taxonomy self.txny ]
birdNoteSet = birdnotes.BirdNoteSet ( txny )
```

3. In the same line as the previous defect, `txny` should be `self.txny`.

```
# [ birdNoteSet := a new birdnotes.BirdNoteSet instance
#     with taxonomy self.txny ]
birdNoteSet = birdnotes.BirdNoteSet ( self.txny )
```

4. The declaration of `MONTH_SEASON_MAP` was cut and pasted from `MONTH_NAME_MAP`, and I changed the content but forgot to change the name.
5. Spelling error in `YearRow.successor()`:

```
def sucesor ( self, mm ):
```

Try:

```
def successor ( self, mm ):
```

6. In `YearCollection.__findNext()`, this line:

```
pos = yyyyList.index ( yyyy )
```

should be:

```
pos = yyyyList.index ( yyyy )
```

7. Method `MonthCell.__pageFrame()` was declared with two arguments `prevURL` and `nextURL`, but the caller was supplying only the “yyyy-mm” key of those months, and the URLs of the target pages were developed inside the method. Change the method's starting lines from:

```
def __pageFrame ( self, prevURL, nextURL ):  
    '''Set up an empty tccpage2.TCCPage instance.  
  
    [ (prevURL is the URL of the previous month page or None) and  
    [ (nextURL is the URL of the next month page or None) ->  
        return a new TCCPage instance with navigation links  
        prevURL and nextURL ]  
    ...
```

to:

```
def __pageFrame ( self, prev, next ):  
    '''Set up an empty tccpage2.TCCPage instance.  
  
    [ (prev is the 'yyyy-mm' of the previous month or None) and  
    [ (next is the 'yyyy-mm' of the next month or None) ->  
        return a new TCCPage instance with navigation links  
        previous=(prev's page) and next=(next's page) ]  
    ...
```

8. Symmetric errors in both `YearRow.firstMonth()` and `YearRow.lastMonth()`: they were supposed to return the month key 'mm', not the entire `MonthCell`. Here is the fix for `YearRow.firstMonth()`:

```
return self[monthKeyList[0]]
```

should be:

```
return self[monthKeyList[0]].mm
```

9. `MonthCell.__renderPage()` was declared as `MonthCell.renderPage()`. This happened during a general renaming of methods of this class so that all the internal methods started with “\_\_”.
10. Left out the `self` argument in `MonthCell.__renderPage()` and also in `MonthCell.__pageTOC()`.
11. In `MonthCell.__pageTOC`, the intended function was correct, but two lines were missing from the code:

```
#-- 1 --  
# [ parent := parent with a new 'ul' child element added  
#   ul := that child  
#   anchorSet := a new, empty set  
#   anchorMap := a new, empty dictionary ]  
ul = et.SubElement ( parent, 'ul' )
```

Here are the missing lines:

```
anchorSet = set()
anchorMap = {}
```

12. In `MonthCell.__dayTOC()`, these lines:

```
newAnchor = self.birdNoteSet.date
suffix = 'a'
while newAnchor in anchorSet:
    newAnchor = self.birdNoteSet.date+suffix
    suffix = chr(ord(suffix)+1)
```

were trying to extract the date from the wrong place. It should be:

```
newAnchor = dayNotes.date
suffix = 'a'
while newAnchor in anchorSet:
    newAnchor = dayNotes.date+suffix
    suffix = chr(ord(suffix)+1)
```

13. In `MonthCell.__dayTOC()`, this line:

```
a = et.SubElement ( 'a', href=('#+newAnchor) )
```

omitted the first argument, and should be:

```
a = et.SubElement ( li, 'a', href=('#+newAnchor) )
```

14. In `MonthCell.__notablesBlock()`, this line:

```
div = et.SubElement ( parent )
```

should be:

```
div = et.SubElement ( parent, 'div' )
```

15. The declaration of `DAY_SUMMARY_CLASS` omitted the actual declaration.

16. The declaration of `MonthCell.__locDef()` was erroneously coded as `.__locDefs()`.

17. In `MonthCell.__paragraph()`, this line:

```
div.addTextMixed ( div, s )
```

should be:

```
tccpage2.addTextMixed ( div, s )
```

18. In `MonthCell.__singleSighting()`, there were four references to `div` that should have been to `parent`.

19. In `MonthCell-ageSexGroup()`, I blithely assumed that `GENUS_CLASS` had already been defined during the coding of `MonthCell.__paragraph()`, but it hadn't: in the latter routine, the class name comes from the input. Solution: define it.

20. In `buildMonthCell()`, omitted the parent argument from `SubElement()`:

```
a = et.SubElement ( 'a', href=monthFileName )
```

should be:

```
a = et.SubElement ( td, 'a', href=monthFileName )
```

21. In `YearRow.predecessor()`, after finding the position of the given month in the list `mmList`, the value `mm` was compared, not its position. So in this code:

```
#-- 3 --  
if mm >= len(mmList):  
    raise KeyError  
else:  
    return mmList[pos+1]
```

the first line should be:

```
if pos+1 > len(mmList):
```

Also, thanks to the wonders of cut'n'paste, `YearRow.successor()` had the same error.

