

# A system for representing bird taxonomy



John W. Shipman

2009-10-30 20:46

## Abstract

Describes a system for representing taxonomic arrangements of bird checklists, and an interface for retrieving such data using the Python programming language.

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 .....	2
2. Files for downloading .....	2
3. Requirements .....	3
4. The six-letter bird code system .....	4
4.1. Design goals for bird code systems .....	4
4.2. Origins of the six-letter code system .....	5
4.3. Rules for the six-letter code system .....	5
4.4. Handling collisions .....	6
5. Input files .....	7
5.1. The ranks file .....	7
5.2. Preparing the standard forms (.std) file .....	8
5.3. Preparing the alternate forms (.alt) file .....	10
6. Building the standard product files .....	13
7. Flat output files .....	13
7.1. The tree (.tre) file .....	13
7.2. The abbreviations (.ab6) file .....	15
7.3. The collisions (.col) file .....	16
8. Schema for the XML product file .....	16
8.1. taxonomySystem: the XML root element .....	16
8.2. rankSet: Taxonomic ranks in use .....	17
8.3. taxonomy: The classification tree .....	18
8.4. abbrSet: Bird code definitions .....	19
8.5. collisionSet: List of collision codes .....	19
9. The Python taxonomy package, txny.py: the interface .....	20
9.1. Class Txny: the complete system .....	20
9.2. Class Hier: The set of taxonomic ranks .....	21

<sup>1</sup> <http://www.nmt.edu/~shipman/xnomo/>

<sup>2</sup> <http://www.nmt.edu/~shipman/xnomo/xnomo.pdf>

9.3. Class Rank: One taxonomic rank .....	22
9.4. The Taxon class: One node in the classification tree .....	23
10. The <code>abbr.py</code> module .....	24
10.1. <code>class BirdId</code> .....	25

## 1. Introduction

---

For the representation and processing of data about wild birds, a firm nomenclatural base is essential. Fortunately, for North American birds, the American Ornithologists' Union (AOU) provides a definitive and inclusive classification of forms that have occurred here, in printed form as the *AOU Check-List*. The AOU's periodical, *The Auk*, periodically publishes supplements updating the names and arrangements.

In the process of entering a few million records of bird data for the Audubon Christmas Bird Count database, the author also developed a system for representing bird names using a six-letter code. This system currently includes over 2500 codes, each code connected to either a taxon or to an English name.

This document describes a software base for representing the AOU's taxonomic arrangements and the author's six-letter codes as computer files. Once the various input files are prepared and compiled, there are several different ways to access these data:

- You can use a single XML file that describes both the taxonomy and the bird code system. Prebuilt files are available for all of the *AOU Check-List* arrangements starting with the Seventh Edition and its various Supplements.
- A module written in the Python programming language makes it easy for you to write scripts in Python that use one of these XML data files.
- You can use a set of three “flat files” that can be loaded into a database system or spreadsheet.

To download any of these data or program files, see Section 2, “Files for downloading” (p. 2).

A companion document, *Bird taxonomy system: internal maintenance specification*<sup>3</sup>, describes the internals of the various programs in this suite.

## 2. Files for downloading

---

Files referred to in this document are available online.

Data files for the *AOU Check-List*, 7th ed., and also for the 42nd-49th Supplements are available online as .zip archives. Each archive contains the input files, the XML output file, and the generated flat files for that revision.

- `aou749.zip`<sup>4</sup>
- `aou748.zip`<sup>5</sup>
- `aou747.zip`<sup>6</sup>
- `aou746.zip`<sup>7</sup>
- `aou745.zip`<sup>8</sup>

<sup>3</sup> <http://www.nmt.edu/~shipman/xnomo/ims2/>

<sup>4</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou749.zip>

<sup>5</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou748.zip>

<sup>6</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou747.zip>

<sup>7</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou746.zip>

<sup>8</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou745.zip>

- [aou744.zip](#)<sup>9</sup>
- [aou743.zip](#)<sup>10</sup>
- [aou742.zip](#)<sup>11</sup>
- [aou7.zip](#)<sup>12</sup>

Python-language source files described herein:

- [nombuild.py](#)<sup>13</sup>
- [xmlcreate.py](#)<sup>14</sup>
- [hier.py](#)<sup>15</sup>
- [abbr.py](#)<sup>16</sup>
- [txny\\_schema.py](#)<sup>17</sup>
- [sysargs.py](#)<sup>18</sup>
- [tree.py](#)<sup>19</sup>
- [set.py](#)<sup>20</sup>
- [scan.py](#)<sup>21</sup>
- [log.py](#)<sup>22</sup>

### 3. Requirements

---

This system was designed originally to provide a nomenclatural infrastructure for data involving North American birds. It was specifically invented to handle Christmas Bird Count census data, but should have general application in other kinds of North American bird records work.

Requirements include:

- The system must cope with the continual changes to the reference taxonomy provided by the American Ornithologist's Union (AOU), allowing multiple taxonomic arrangements corresponding to the successive versions of *The A.O.U. Check-list of North American Birds*<sup>23</sup>.
- It must provide a way of sorting bird records into *phylogenetic order*, the traditional way of arranging bird taxa.

To satisfy this requirement, the system provides a *taxonomic key number* for each taxon in an arrangement. Sorting records on this key puts things in phylogenetic order.

<sup>9</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou744.zip>

<sup>10</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou743.zip>

<sup>11</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou742.zip>

<sup>12</sup> <http://www.nmt.edu/~shipman/xnomo/aou/aou7.zip>

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

<sup>14</sup> <http://www.nmt.edu/~shipman/xnomo/xmlcreate.py>

<sup>15</sup> <http://www.nmt.edu/~shipman/xnomo/hier.py>

<sup>16</sup> <http://www.nmt.edu/~shipman/xnomo/abbr.py>

<sup>17</sup> [http://www.nmt.edu/~shipman/xnomo/txny\\_schema.py](http://www.nmt.edu/~shipman/xnomo/txny_schema.py)

<sup>18</sup> <http://www.nmt.edu/~shipman/xnomo/sysargs.py>

<sup>19</sup> <http://www.nmt.edu/~shipman/xnomo/tree.py>

<sup>20</sup> <http://www.nmt.edu/~shipman/xnomo/set.py>

<sup>21</sup> <http://www.nmt.edu/~shipman/xnomo/scan.py>

<sup>22</sup> <http://www.nmt.edu/~shipman/xnomo/log.py>

<sup>23</sup> <http://www.aou.org/checklist/index.php3>

- It must support a system of short codes for bird names. The code system must be easy to learn and use. Codes must be short yet meaningful and easily translated back to names.

## 4. The six-letter bird code system

---

There are a number of different systems for encoding kinds of birds. This package supports a particular system that grew out of the author's work creating a database of 90 years of Christmas Bird Count data. The package could be modified to work with other systems.

### 4.1. Design goals for bird code systems

Among programmers, good programs or systems are often described as “robust.” Such systems should be easy to learn and use, and they should not tend to confuse users or mangle data. Design of a good encoding system involves more than just the problem of representing the data. We should consider human factors as well.

Here are some other qualities of a good code system:

- Codes should be *short*, to save keystrokes during data entry.
- *Encoding* should be easy to learn and quick to execute.
- The codes should be meaningful and easy to *decode*. Although any code can be translated mechanically by a program, it often saves time if we can just look at a code and know what it means without having to look it up.
- It should handle *forms other than species*— any category of birds, however precise (“Blue Goose”) or vague (“black bird sp.”) the identification.
- It should cope well with the *continual changes* in taxonomy and nomenclature.
- It should be usable even by *non-experts*, so beginners and even non-birders can use it for data entry.
- Use of a code should not be a significant *source of errors*.

For efficient data entry, we want to be able to bang the records into the machine quickly (minimizing mistakes, of course). Speed depends on more than just the keystroke rate. Thinking takes time too—the time it takes to think of the right code, or look it up if necessary.

A robust system should also be designed so that most errors can be detected easily, and easily corrected whenever possible. In the author's opinion, this is an argument against using the shortest possible code. Longer codes have more redundancy, so it is more likely that a user can figure out what was meant even if the code has an error in it. As an example, the English language has a lot of redundancy in it, which is a robust characteristic. We can oftentimes understand a sentence if it contains quite a few typos.

The best way to represent the name of a bird is to spell it out, and to conform (where possible) to the names standardized in the current edition of the *AOU Check-List*.

However, using the AOU standardized names has some drawbacks:

- For computer applications requiring bird names to be encoded for storage, typing a full name is prohibitively inefficient.
- The AOU does not address the common problem of representing imprecise identifications such as “duck sp.” or “large falcon.”
- A system for retrieval of sight records must be able to handle exotics (e.g., escaped waterfowl) that are not included in the *AOU Check-List*.

## 4.2. Origins of the six-letter code system

The new coding system presented here was invented by the author for use in preparing a database of Christmas Bird Count (CBC) data. We found that using the banders' four-letter code to enter data was very frustrating, as we spent far too much time consulting the list of exceptions.

After some experimentation, we found a six-letter system to be a good tradeoff. Even though each code takes two extra keystrokes, the number of special cases went down by an order of magnitude. This greatly reduces thinking time, making the work flow more smoothly, and significantly increases the throughput measured in records per hour.

We have used these codes to enter over three million historical CBC records: all the North American and Hawaiian counts from the 1st CBC, in 1900, through the 90th CBC in 1989 (except for ten years, the 63rd-72nd counts, which were provided by Dr. Carl Bock's project at the University of Colorado). As an example, the 89th CBC, with 1523 count circles and about 115,000 records, took less than 50 hours to enter at a keying rate of about 65 words per minute. This translates to a rate of well over 2,000 records per hour.

Many refinements in this system were suggested by Greg Butcher and Jim Lowe of the Cornell Lab of Ornithology. The author greatly appreciates their contributions.

## 4.3. Rules for the six-letter code system

As with the Bird Banding Lab's four-letter codes, six-letter codes are derived by abbreviating the name of the bird. Names are not limited to standard AOU species names. Codes may be based on obsolete names (e.g., Short-billed Marsh Wren), subspecies names (Peale's Falcon), color morphs (Blue Goose), or even vague categories like "raptor" or "*Empidonax* sp."

1. Birds with one-word names are abbreviated by taking the initial letters of the name:

CANVAS	Canvasback
RUFF	Ruff
MURREL	murrelet sp.
EMPIDO	<i>Empidonax</i> sp.

2. For two-word names, take the first three letters of the first word and the first three letters of the last word. Hyphenated words are always treated as separate words:

CEDWAX	Cedar Waxwing
LARFAL	large falcon sp.
STOPET	storm-petrel sp.

3. For three-word names, take two letters from the first word, one from the second, and three from the third:

BABWAR	Bay-breasted Warbler
GRPCHI	Greater Prairie-Chicken
DABSHE	dark-backed shearwater sp.

4. For four or more words, take one letter each from the first three words, then the first three letters of the last word:

GBBGUL	Great Black-backed Gull
BCNHER	Black-crowned Night-Heron
BTBWAR	Black-throated Blue Warbler

5. To reduce the number of conflicts, certain similar color names are abbreviated in standard ways:

Color name	Three-letter form	Two-letter form
green	GRN	GN
gray	GRY	GY
black	BLK	BK
blue	BLU	BU
brown	BRN	BN

Examples:

BLKPHO	Black Phoebe
GRYJAY	Gray Jay
GNWTEA	Green-winged Teal

## 4.4. Handling collisions

By *collision*, we mean a case where two or more names would have the same code while applying the rules. For example, Blackburnian Warbler and Blackpoll Warbler would both be encoded as BLAWAR. In that case we devise substitute codes that still suggest the original names: in this case, BKBWAR for Blackburnian Warbler and BKPWAR for Blackpoll Warbler.

### Warning

We feel very strongly that in all collision cases, the “collision code” **must not be used**. This allows automatic detection of encoding errors by computer programs. Having a computer detect an error takes a lot less time than catching it in proofreading, assuming you have time to proof it at all.

Therefore, code BLAWAR, and the other collision forms, are not allowed, even in cases where one of the names is extremely unlikely to occur. For example, American Goldfinch collides with an ancient and obsolete name for Common Goldeneye, “American Goldeneye.” We disallow the collision code AMEGOL and require the use of the substitute codes AMEGFI and AMEGEY. This is a necessity in the Christmas Bird Count database because so many old names occur in the early days of that census.

The author's long career working with banding data has convinced him that the four-letter codes devised by the US Fish & Wildlife Service's Bird Banding Lab are highly error-prone. Roughly 100 names are involved in collisions in this system!

Most banders work with a fairly small set of local species: a bander on the East Coast may encoded Carolina Wren as CAWR, not realizing that this code is a three-way collision with Cactus Wren and Canyon Wren. When the code CEWA shows up on a banding sheet, it is not always clear whether it

means Cerulean Warbler or Cedar Waxwing. The BBL also generally does not use the collision form. In this case they supply substitute codes CERW and CEDW, so that the appearance of code CEWA on a banding sheet is always an error.

However, they violate this rule in a few cases where one of the colliding names is highly unlikely to occur on a banding sheet. For example, because Barnacle Goose is quite rare, code **BAGO** is allowed for Barrow's Goldeneye, with code **BRNG** used for Barnacle Goose. The author feels that this is risky, since it assumes that everyone working with bird records knows something about bird distribution.

## 5. Input files

---

This system supports multiple taxonomic arrangements. Each arrangement is represented as a pair of input files: one describing the standard names and arrangement from a specific edition of the *AOU Check-List*, and a companion file describing all the other names that may occur.

The author prefers to maintain the source files for taxonomic arrangements as simple text files with a modest structure. This simplifies the preparation of original data files, as well as making them easier to maintain when new AOU checklists are made available. At this writing, files are available for all arrangements starting with the published *AOU Check-List*, Seventh Edition, proceeding through all the biennial supplements through the *Forty-fifth Supplement* published in 2004.

Each taxonomic arrangement is represented as a set of three text files:

- The *ranks file* describes the set of taxonomic ranks (or levels or aggregates) of interest to the applications programs. The *AOU Check-List* describes a number of taxa such as subclasses and superorders that are not usually of interest, so those aggregates can be omitted from the ranks file. If different applications have different needs, they can provide different versions of the ranks file. For example, some applications might wish to use subfamily rank, while other applications might not.
- The *standard forms file* describes all the *standard* taxa, that is, all those in the approved arrangement. A typical source for this file is the *AOU Check-List*.
- The *alternate forms file* enumerates all names and identifiable forms other than the standard taxa. Some are obsolete names (e.g., rather than "Cardinal," the preferred name is now "Northern Cardinal"). Some are names for proper subsets of species (e.g., races or color morphs like Eurasian Green-winged Teal or Blue Goose). Some are names for larger aggregates (e.g., "hawk spp.").

The following sections describe the format of these raw files. Later sections will discuss the process of building the product files to be used by applications.

### 5.1. The ranks file

The *AOU Check-List* defines a lot more taxonomic ranks than most applications will care about, so the *ranks* file allows the application to specify which ranks are of interest. To prepare this file, use a text editor to enumerate the ranks in descending order, starting with the rank of the root taxon of the arrangement.

Each line defines one rank. Enter these items in order:

1. A two-character code for the taxonomic rank, such as "-f" for Subfamily. If the code is only one character long (e.g., "f" for family), it should be placed in the first column, with a space in the second column.
2. In the third column, put a space if the rank is mandatory, that is, if every lower taxon *must* be placed in such a rank.

For ranks that are not always used (such as Subfamily in the *AOU Check-List*), enter a question mark (“?”) in this column to indicate that the rank is optional. Because they are the bedrock of the binomial system, *genus and species ranks may not be optional*.

3. In the fourth column, specify the number of digits to be allocated in the taxonomic key for this rank. This value should be “1” if there are never more than 9 of this taxon in the next higher one; “2” if there are no more than 99; and so on. *The first (root) taxon must have a value of “0”, or your taxonomic keys will have a useless, always-one prefix.*
4. On the remainder of the line, enter the name of the taxonomic rank, such as “Genus”.

Here is a sample `ranks` file.

```
c 0Class
o 2Order
f 2Family
-f?1Subfamily
g 2Genus
s 2Species
x 2Form
```

The numbers in this example allow for up to 99 orders per class, 99 families per order, 9 subfamilies per family, and so on.

The last three lines of this file use codes that do not actually appear in the standard forms file:

- Code `g` is used for genus.
- Code `s` is used for species.
- Code `x` is used for identifiable forms that are subsets of species, such as races or morphs.

Applications that don't need to track subspecific forms should use a version of the `ranks` file that omits the `x` line. Omitting the `g` and `s` lines is not recommended.

The order is important---always order the ranks from largest to smallest, as in the example above. The program doesn't know anything about taxonomic traditions. If you would like to create your own new ranks, like Infrasertribes, go right ahead.

## 5.2. Preparing the standard forms (`.std`) file

The first input file you must prepare is the standard forms file. This file enumerates all the taxa defined in your preferred standard arrangement. Give this file a name of the form `f.std` where `f` is some name suggesting the source of the arrangement. For example, a file containing names from the *AOU Check-List*, 6th ed., including all supplements through the 40th, might be named `aou640.std`.

Place each taxon on a separate line in the standard forms file. The taxa appear in the order in which they are presented in a checklist. The highest taxon appears first, followed by the first contained taxon, and so on down to the first species. The remaining species in that genus follow; then come the other genera in the family, and so on.

Some taxa are defined implicitly. In particular, there is no separate line for genera, since species are identified by binomials: genera are declared implicitly by their first use in a binomial.

There are two types of record in the standard forms file:

- Each *higher taxon record* represents a taxon above the generic rank. All such records start with a non-blank character.

- Each *species record* represents a single species in the standard checklist. These records each start with a space.

Records in the standard forms file start with three fixed columns, with the remainder of the record in a variable-length format:

1. The first two columns are the code for the taxonomic rank. Any one- or two-character code may appear here, but one-letter codes must be left-justified and padded with a space. Here are the codes used for one representation of the *AOU Check-List*:

c	Class
- c	Subclass
+o	Superorder
o	Order
- o	Suborder
+f	Superfamily
f	Family
- f	Subfamily
t	Tribe
(blank)	Species

2. The third column defines the status of the bird. This column is normally blank, but can contain a status code: “?” to indicate that the species is of questionable occurrence in this checklist, or “+” for species that are extinct.

The exact structure of the “tail” of the record (that is, the variable-length part that follows the first three columns) depends on whether the record describes a higher taxon or a species.

### 5.2.1. Higher-taxon records in the .std file

Place each higher taxon on a separate line, following these steps:

1. Type the two-letter rank code, as defined in the *ranks* file (see the section above). If the code has only one letter, enter the letter followed by one space.
2. In the third column, enter the status code. This is usually one space, but encode it as “?” for dubious taxa or “+” for extinct taxa.
3. Enter the scientific name of the taxon.
4. Type one slash (“/”), followed by the English name of the taxon.

Here are some examples of higher-taxon records:

```
c  Aves/Birds
-c Neornithes/True Birds
+o Neognathae/Typical Birds
o  Gaviiformes/Loons
f  Gaviidae/Loons
-o Pelecani/Boobies, Pelicans, Cormorants and Darters
```

### 5.2.2. Species records in the .std file

For each species record, enter these fields on one line:

1. Place two spaces in the first two columns.
2. Enter the one-character status code in the third column. This is normally blank, but may be “?” for dubious or “+” for extinct.
3. Enter the scientific name. The taxon is generally a binomial, but it may include a subgenus as it customarily represented: in parentheses, between the genus and species names.
4. Enter one slash (“/”), followed by the English name. You may enter multi-word names either in the conventional order (e.g., “Wood Duck”), or with the generic part first, followed by a comma and the specific part (e.g., “Duck, Wood”).
5. In most cases, you're done. However, if this species is involved in a collision—that is, if it is one of a group of two or more names that abbreviate to the same code according to the rules—enter another slash (“/”) followed by the *disambiguation*, that is, the substitute code for this species.

Here are some examples of species lines. The last two show the disambiguation of the collision for code BLAWAR.

```
Anas strepera/Gadwall
Anas penelope/Wigeon, Eurasian
Haliaeetus pelagicus/Sea-Eagle, Steller's
+Camptorhynchus labradorius/Labrador Duck
?Aerodramus vanikorensis/Gray Swiftlet
Cygnus (Olor) buccinator/Trumpeter Swan
Cygnus (Cygnus) olor/Swan, Mute
Dendroica fusca/Warbler, Blackburnian/BKBWAR
Dendroica striata/Blackpoll Warbler/BKPWAR
```

### 5.3. Preparing the alternate forms (.alt) file

Because field records do not always use the latest names, and because the reported forms are not always standard species, you must prepare an “alternate forms” file enumerating all the forms that have a six-letter code but which are not standard species names.

You must prepare an .alt file for each .std file, reflecting the exact lumps, splits, and names of the standard arrangement. The file must be named *f*.alt, where *f* is the same prefix as that of the .std file.

For example, if the standard file for the *AOU Check-List*, 6th. ed., including supplements through the 40th, is called `aou640.std`, the corresponding alternate names file must be called `aou640.alt`.

In the .alt file you will place several different types of records. Each line starts with the six-letter code being defined, followed by a *record type code*, and a variable length *tail*.

#### 5.3.1. Higher taxon records in the .alt file

For each form above species rank in the hierarchy, enter a line of this format:

1. Enter the six-letter code. If the code is shorter than six letters (e.g., HAWK), right-pad it to length with spaces.
2. Enter one space. This signifies that the record is for a higher taxon.

3. Enter the scientific name of the higher taxon to which this code is referred. This name must be defined in the `.std` file.
4. Enter one slash (“/”), then the English name.
5. In most cases, you are done. However, if the English name requires some markup to be represented correctly in typeset output, enter another slash, followed by the English name formatted according to the T<sub>E</sub>X typesetting system.

In the optional T<sub>E</sub>X name field, two T<sub>E</sub>X macros are used:

- The `\sp` macro takes one argument and formats it in italic followed by “sp.” in Roman type. Here is the T<sub>E</sub>X definition of this macro:

```
\def\sp#1{\itc{#1} sp.}%
```

- The `\itc` macro formats its argument in italics, followed by the italic correction (`\/`). Here is its definition:

```
\def\itc#1{{\it #1\/}}%
```

Here are some complete examples of higher-taxon records.

```
albatr Diomedidae/albatross sp.
accipi Accipiter/Accipiter sp./\sp{Accipiter}
laracc Accipiter/large Accipiter sp./large \itc{Accipiter} sp.
```

### 5.3.2. Direct equivalent records in the `.alt` file

For each non-standard code that is the exact equivalent of a standard code, create a record in the alternate forms file with this format:

1. Enter the non-standard code, left-justified in the first six columns.
2. Enter an equal sign (=) in the seventh column. This is the record type code for an exact equivalent.
3. Enter the standard six-letter code for the new name, left-justified in the next six columns.
4. Enter one space, followed by the English name (for annotation purposes).

Examples of direct-equivalent records:

```
amboys=blkoys Oystercatcher, American Black
amewid=amewig Widgeon, American
watpip=amepip Pipit, Water
```

#### Note

The form after the equal sign must be defined elsewhere in the standard or alternate forms file.

### 5.3.3. Subspecific forms records in the `.alt` file

There are several reasons for assigning codes to forms that are a subset of a standard species:

- Subspecies in the strict taxonomic sense, such as Myrtle Warbler (a subspecies of Yellow-rumped Warbler).

- Color morphs, such as Blue Goose (a morph of Snow Goose).
- Recognizable forms of uncertain taxonomic status, such as Pink-sided Junco (an identifiable form of Dark-eyed Junco).

So we use the term “subspecific form” loosely, to mean any identifiable form that refers to some subset of a standard species. For each such code, enter a line with this format:

1. The six-letter code being defined.
2. A less-than (<) symbol. This is the record type code for a subspecific form record.
3. The six-letter code of the standard species that contains this form.
4. One space, followed by the English name of this form.
5. In most cases you are done. However, if the English name needs T<sub>E</sub>X markup to appear correctly in typeset output, append a slash, followed by the T<sub>E</sub>X-encoded English name.

Examples of subspecific form lines:

```
agpchi<grpchi Attwater's Greater Prairie-Chicken
agwtea<gnwtea Teal, American Green-winged
alcgoo<cangoo (Aleutian) Canada Goose
axetea<gnwtea teal, (American x European) Green-winged
blugoo<snogoo Blue Goose
branth<brant Brant (hrota)/Brant (\itc{hrota})
```

### 5.3.4. Collision records in the .alt file

In order to record all the known collisions—that is, cases where two or more names encode to the same six-letter abbreviation according to the rules for abbreviation formation—you must add to the alternate forms file one line for each collision. Each such line enumerates all the disambiguations, that is, the substitute form codes that are preferred:

1. Enter the collision code in the first six columns.
2. Enter a question mark (?) in the seventh column.
3. Type all the disambiguations separated by colon (:) characters.

Examples of collision records:

```
barowl?brdowl:brnowl
belspa?bldspa:blspa
columb?colba :colbid:colbin
```

The first example shows that two names collide for the code `barowl`. The forms are Barred Owl (which is given the substitute code `brdowl` in the standard forms file) and Barn Owl, with substitute code `brnowl`.

The last example shows a three-way collision for code `columb` between the codes for genus *Columba*, family Columbidae, and subfamily Columbinae. Note that a collision record may refer to forms other than standard taxa.

## 6. Building the standard product files

---

Once you have prepared all the input files, you can compile them into a set of standard product files. There are two forms of output:

- One convenient form is an XML file that incorporates all the various facets of the taxonomy into a single file. XML tools are becoming quite common nowadays, so this file may be sufficient for your needs. The structure of this file is described below using the Relax NG schema language; see Section 8, “Schema for the XML product file” (p. 16).

This file is also the only input necessary for the Python-language interface described later in this document; see Section 9, “The Python taxonomy package, `txny.py`: the interface” (p. 20).

- The same information can instead be written to a set of “flat files” that give the same information in a form more immediately usable in database applications. See Section 7, “Flat output files” (p. 13).

The `nombuild` program checks the various input files and writes its output as either XML or flat files. To run this program, change to the directory containing all the input files and type the command in this format:

```
nombuild [-x] basename
```

where the command line options are:

**-x**

Use this option to write XML output files. The default is to write the output as a set of flat files.

***basename***

This is the name of the standard forms file without its “.std” suffix. The program expects to find a corresponding alternate forms file with the same base name and suffix “.alt”.

If there are any problems with the input files, the program will produce error messages on the standard output stream, and also produce a duplicate listing of these errors in file `nombuild.log`.

## 7. Flat output files

---

If you are planning on representing bird taxonomy using a relational database, flat files are a universal format accepted by all the major database systems. In a flat file, each record consists of a sequence of fixed-length fields.

If you run the `nombuild` program without the `-x` option, three files are written by that program:

- The *tree file* defines all the taxa in the standard forms file plus all subspecific taxa from the alternate forms file. Its name is the same as the input file, except it has extension `.tre`. For example, if the input files are `aou640.std` and `aou640.alt`, the tree file will be called `aou640.tre`.
- The *abbreviations file* defines all the six-letter bird codes. This file has extension `.ab6`.
- The *collisions file* describes every six-letter bird code that is invalid because two or more names would all abbreviate to that code. Its extension is `.col`.

The sections below describe the formats of these product files.

### 7.1. The tree (`.tre`) file

The tree file defines all the different scientific names used in the input. Here is the format of that file:

Length	Contents
varies	The taxonomic key number. The exact format of this field depends on the content of the <code>ranks</code> file; see Section 7.1.1, “Taxonomic key numbers” (p. 14).
6	If this taxon has a standard six-letter bird code, that code appears here; otherwise the field is blank.
1	For generally accepted forms, this field is blank. If the form is not in the main <i>AOU Check-List</i> , a question mark (?) appears here.
36	The next field is the scientific name of the group to which this form is referred, for example, <i>Junco hyemalis</i> . The field is aligned flush left and padded on the right with spaces. For forms not identified to species, the smallest containing taxon is used, e.g., <i>Aves</i> for “bird sp.”  For subspecific forms defined in the alternate names file, this field contains the scientific name with a space and an integer appended. For example, in the line for the standard species Snow Goose, this line will have the value “ <i>Chen caerulescens</i> ”, while Blue Goose will have “ <i>Chen caerulescens</i> 1”, Blue-Snow intergrade “ <i>Chen caerulescens</i> 2”, and so on.
56	The English name of the form appears next, aligned flush left and right-padded with spaces. For multi-word names, the generic part comes first, followed by a comma, one space, and the specific part.  Examples:  <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> Dunlin  Loon, Red-throated  grebe sp.  bird sp.  bird, large sp.  teal, Blue-winged x Cinnamon  Junco, (Gray-headed x Slate-colored) Dark-Eyed </div>
varies	At the end of the record is a variable-length field containing the English name, encoded for typesetting using T <sub>E</sub> X markup codes. Use this field to get diacritical marks and correct italicization of generic names.

### 7.1.1. Taxonomic key numbers

The taxonomic key number can be used to sort records into phylogenetic order, as defined by the *AOU Check-List*. It contains one or more digits for each rank (except for the root rank). The number of digits for each rank is determined by the third column in the `ranks` file.

#### Warning

It is an extremely bad idea to use this number to represent a taxon for any other purpose other than sorting. Not only is it spectacularly meaningless out of context, but any change to the input files will change all of the taxonomic key numbers.

For example, if your `ranks` file looks like the example given above (2-digit order, 2-digit family, 1-digit subfamily, 2-digit genus, 2-digit species, and 2-digit form), each taxonomic key number would have these components:

- The two-digit serial number of the taxonomic order in which this form is placed, or “00” if the form is not placed into an order (e.g., “bird sp.”).

- The two-digit serial number of the taxonomic family within this order, or “00” for forms not placed within a specific family. Note that the sequence of families starts over at “01” again within each order.
- The one-digit serial number of the subfamily within the family, or “0” if the subfamily is unknown.
- The two-digit serial number of the genus within the family, or “00” if the genus is unknown.
- The two-digit serial number of the species within the genus, or “00” if the species is unknown.
- The two-digit serial number of the form within the species, or “00” if the form is unknown.

For example, code `daejun` (Dark-eyed Junco) might have a taxonomic key number of “21 24 3 47 01 00” (the spaces here are for clarity—they are not actually present in the record). This key would mean that this form is in the 21st order, and in the 24th family within that order, the 3rd subfamily within that family, the 47th genus within that subfamily, and the first species within that genus, and not in any known subform of the species.

Other forms that are included within Dark-eyed Junco will have keys “21 24 3 47 01 01”, “21 24 3 47 01 02”, and so on. Examples of such forms include races such as Gray-headed Junco, hybrids among the different races (e.g., “Gray-headed × Slate-colored Junco”), and obsolete names (“Northern Junco”).

Note that the taxonomic key number can be used to deduce relationships between form codes. For example, to find out what genus a species is in, just construct a key number that is the same as the species’ key number, but with its species number set to “00”. Continuing the example above, suppose Gray-headed Junco has this key number:

```
21 24 3 47 01 01
```

Then we can deduce all the higher ranks by substituting zeroes in the appropriate fields:

21 24 3 47 01 00	The containing species, <i>Junco hyemalis</i>
21 24 3 47 00 00	The containing genus, <i>Junco</i>
21 24 3 00 00 00	The containing subfamily, Emberizinae
21 24 0 00 00 00	The containing family, Emberizidae
21 00 0 00 00 00	The containing order, Passeriformes
00 00 0 00 00 00	The containing class, Aves

## 7.2. The abbreviations (.ab6) file

The `.ab6` file defines all the six-letter bird abbreviations. Each abbreviation is specified by its `taxon` field, which is a relational link to the corresponding taxon record in the tree file. Fields are:

Length	Contents
6	The six-letter bird code, uppercased and aligned flush-left in the field with right blanks.
36	The taxon to which this code is referred; a relational link to the same field in the <code>.tre</code> file.
varies	The English name from which this abbreviation was derived, with no trailing whitespace.

Here are examples of lines from an `.ab6` file:

```
ALCG00Branta canadensis 2      (Aleutian) Canada Goose
CALLINCarpodacus mexicanus     California Linnet
```

The first is for code ALCG00, derived from the name “Aleutian Canada Goose,” and it is the second subspecific form for *Branta canadensis*, the Canada Goose. The second line is for code CALLIN, derived from the name “California Linnet,” an alternate name for House Finch, *Carpodacus mexicanus*.

### 7.3. The collisions (.col) file

The .col file enumerates all the six-letter form codes that are involved in collisions. Each line has this format:

Length	Contents
6	The collision code, invalid because two or more names would abbreviate to that code by the rules.
6	One of the valid codes that has been substituted for the collision code.

Here is an example showing three records from a .col file. These three lines document the collision between three names for code PASSER. The preferred substitute codes are PASINA (for *Passerina*), PASINE (for “passerine”), and PASR (for *Passer*):

```
PASSERPASINA
PASSERPASINE
PASSERPASR
```

## 8. Schema for the XML product file

The format of a data file that uses the XML (Extended Markup Language) syntax must be described by a *schema*, a formalized description of a particular document type. The format of the XML file written by the nombuild program is described here using the Relax NG schema language in its compact syntax (RNC). For more information on RNC, see Relax NG Compact Syntax (RNC)<sup>24</sup>.

### 8.1. taxonomySystem: the XML root element

The starting symbol for this schema is `taxonomySystem`, a container for the entire file:

```
txny.rnc
```

```
# Relax NG schema for bird code and taxonomy system.
#   For documentation, see:
#   http://www.nmt.edu/~shipman/xnomo/
#
start = taxonomySystem
taxonomySystem = element taxonomySystem
{ attribute date { text }?,
  rankSet,
  taxonomy,
  abbrSet,
  collisionSet
}
```

#### date

This optional attribute can hold an RCS date tag or other automatic modification timestamp.

<sup>24</sup> <http://www.nmt.edu/tcc/help/pubs/rnc>

**rankSet**

The `rankSet` child element holds a list of the taxonomic ranks used in the classification.

**taxonomy**

The `taxonomy` child element contains the entire classification as a taxonomic tree structure.

**abbrSet**

All the valid six-letter codes used in the system, whether standard or alternate codes, are defined under the `abbrSet` child element.

**collisionSet**

This child element contains a list of the six-letter codes that are invalid because they are collisions.

## 8.2. rankSet: Taxonomic ranks in use

The `rankSet` element is a simple container for `rank` elements, each of which defines one taxonomic rank. The ranks must be in descending order of size, starting with the rank of the root taxon of the classification.

txny.rnc

```
rankSet = element rankSet { rank+ }
rank = element rank
{ attribute code { text },
  attribute optional { '1' }?,
  attribute digits { xsd:string { pattern='[0-3]' } },
  attribute depth { xsd:nonNegativeInteger },
  text
}
```

**code**

This is the rank code as described under Section 5.1, “The ranks file” (p. 7). Examples: “o” for order, “-f” for subfamily.

**optional**

Attribute `optional='1'` means that this rank may not appear in some parts of the classification.

**digits**

Specifies the number of digits for this rank in the taxonomic key. See Section 7.1.1, “Taxonomic key numbers” (p. 14).

**depth**

The first or root rank will have `depth='0'`, and each successively deeper rank will have a value that is one greater than its predecessor. This value can be used to answer the question, which rank is deeper?

**text**

The textual content of the `rank` element is the name of the rank, e.g., “Genus”.

Here's an example of the `rank` element for subfamily rank:

```
<rank code='-f' optional='1' digits='1' depth='3'>
  Subfamily
</rank>
```

### 8.3. taxonomy: The classification tree

The **taxonomy** element is a container for a tree composed of **taxon** elements, describing the arrangement of taxa in the biological classification. The **taxonomy** element has exactly one **taxon** child, representing the root taxon of the classification (Class Aves, for birds). That **taxon** element has child **taxon** elements representing the orders, each order element has child elements representing the families, and so on down to the lowest-level ranks.

txny.rnc

```
taxonomy = element taxonomy { taxon }
taxon = element taxon
{ attribute rank { text },
  attribute status { ' ' | '?' | '+' }?,
  attribute stdAbbr { text }?,
  attribute sci { text },
  attribute txKey { text },
  eng,
  texName?,
  taxon*
}
eng = element eng { text }
texName = element texName { text }
```

#### **rank**

The rank code for this taxon, e.g., “rank= 's' ” for a species.

#### **q**

The optional q= '1' attribute indicates that the taxon is questionable.

#### **stdAb**

If the taxon is a species or form, this attribute gives its standard six-letter bird code.

#### **sci**

The taxon's scientific name, e.g., “sci= 'Haliaeetus leucocephalus' ”.

#### **txKey**

The taxonomic key number for this taxon. See Section 7.1.1, “Taxonomic key numbers” (p. 14).

#### **text**

The text content of a **taxon** element is the English name of the form, devoid of any markup for italics. Capitalization follows the *AOU Check-List*.

#### **texName**

The **texName** child element contains the English name, marked up using T<sub>E</sub>X markup conventions, as described above under Section 5.3, “Preparing the alternate forms (.alt) file” (p. 10).

#### **taxon\***

Each **taxon** element can have zero or more **taxon** child elements describing contained subtaxa.

Here's an example of a complete **taxon** element at the bottom of the tree:

```
<taxon rank='s' stdAb='BOBOLI' sci='Dolichonyx oryzivorus'
  txKey='22350010100'>
  Bobolink
</taxon>
```

## 8.4. abbrSet: Bird code definitions

The `abbrSet` element is a simple container for `abbr` elements, each of which defines one of the valid six-letter bird codes.

txny.rnc

```
abbrSet = element abbrSet { abbr* }
abbr = element abbr
{ attribute code { text },
  attribute sci { text },
  text,
  texName?
}
```

### code

The `code` attribute of an `abbr` element is the bird code being defined, in uppercase.

### sci

The `sci` attribute is the scientific name of the taxon to which this code is assigned. All such attribute values must be defined in some child of the `taxonomy` element.

### text

The text content of an `abbr` element is the English name from which that code was derived.

### texName

The `texName` child element contains the English name, marked up using the usual T<sub>E</sub>X conventions.

Here's an example of an `abbr` element:

```
<abbr code='CALLIN' sci='Carpodacus mexicanus'>
  California Linnet
</abbr>
```

## 8.5. collisionSet: List of collision codes

The `collisionSet` element is a simple container for `collision` elements, each of which describes one cluster of codes involved in a collision.

txny.rnc

```
collisionSet = element collisionSet { collision* }
collision = element collision
{ attribute badAbbr { text },
  goodAbbr+
}
goodAbbr = element goodAbbr { text }
```

The invalid code is the `badAbbr` attribute, and the `collision` element has one `goodAbbr` element for each of the valid substitute codes involved. Example:

```
<collisionSet>
  <collision badAbbr='ALSSPA'>
    <goodAbbr>ALASPA</goodAbbr>
    <goodAbbr>ALESPPA</goodAbbr>
    <goodAbbr>ALMSPA</goodAbbr>
  </collision>
</collisionSet>
```

```
...
</collisionSet>
```

## 9. The Python taxonomy package, `txny.py`: the interface

The author considers the Python language the best general-purpose programming currently available. This section describes a Python-language module suited for putting a firm taxonomic foundation under bird records work.

Python module `txny.py` provides access to an XML taxonomy file as described above under Section 8, “Schema for the XML product file” (p. 16). It insulates you from the XML files, providing attributes and methods that allow you to look up bird codes and other common operations in bird records management.

To use this module, import it like this:

```
from txny import *
```

Here are the exported classes available in the `txny.py` module.

### 9.1. Class `Txny`: the complete system

Normally the first thing you'll do is instantiate a `Txny` object, which represents the entire system—taxonomy and bird codes:

#### **`Txny ( dataFile=None )`**

Reads the XML data file and returns a `Txny` object representing that file.

If no argument is supplied, this constructor looks for a file named `aou.xml` in the current directory.

If there is no readable, valid XML file, the constructor raises an `IOError` exception.

Attributes of a `Txny` object include:

#### **`.root`**

The root taxon of the taxonomic arrangement, as a `Taxon` object. See Section 9.4, “The `Taxon` class: One node in the classification tree” (p. 23).

#### **`.hier`**

Contains a `Hier` object representing the set of taxonomic ranks used. See Section 9.2, “Class `Hier`: The set of taxonomic ranks” (p. 21).

Methods on a `Txny` object include:

#### **`.lookupTxKey ( txKey )`**

Looks for the taxon corresponding to taxonomic key number `txKey` and returns it as a `Taxon` object. Raises a `KeyError` exception if the arrangement has no such key.

#### **`.lookupSci ( sci )`**

Looks for the taxon whose scientific name matches `sci`, and returns it as a `Taxon` object. This is a case-sensitive comparison. If there is no taxon in this arrangement with the given scientific name, raises `KeyError`.

#### **`.lookupAbbr ( abbr )`**

Looks for the taxon that is equivalent to bird code `abbr`, and returns it as a `Taxon` object.

### **.lookupCollision(abbr)**

If `abbr` is one of the bird codes disallowed because it is a collision, this method returns a list of the valid substitute codes. For example, this call

```
txny.lookupCollision("BAROWL")
```

would return the list `["BRDOWL", "BRNOWL"]`.

Raises `KeyError` if `abbr` is not a collision code.

### **.genTxKeys()**

Generates the taxonomic keys in the arrangement in ascending (phylogenetic) order, as strings. In case you are not familiar with Python generators, a relatively new language feature, see Python 2.2 quick reference<sup>25</sup> under the section "Recent features."

### **.genAbbrs()**

Generates the valid bird codes in self, in ascending order, uppercased.

### **.abbrToEng(abbr)**

Returns the English name from which the given abbreviation `abbr` was derived. Raises `KeyError` if `abbr` is not valid.

### **.abbrToTeX(abbr)**

Returns the English name from which the given abbreviation `abbr` was derived, marked up for TeX. If there is no specific TeX markup, it returns the English name as regular text. Raises `KeyError` if `abbr` is not valid.

Here's a brief example. Assume you have a taxonomy file named `aou.xml` in your directory, and you want to print out the scientific name and English name corresponding to the code 'GOCKIN'. This code would do the trick:

```
from txny import *

txny = Txny ( 'aou.xml' )
taxon = txny.lookupAbbr ( 'gockin' )
print "%s [%s]" % (taxon.eng, taxon.sci)
```

This will print the line:

```
Golden-crowned Kinglet [Regulus calendula]
```

## **9.2. Class Hier: The set of taxonomic ranks**

Once you have instantiated a `Txny` object, that object has a `.hier` attribute that is an instance of class `Hier`, which represents the set of taxonomic ranks or levels in use for this arrangement.

Attributes of the `Hier` object include:

### **.txKeyLen**

Length of the taxonomic key number string used in this arrangement, as an integer. Read-only.

Methods on a `Hier` object include:

### **.canParentHaveChild(p, c)**

This predicate tests whether a rank `p` can have an immediate child `c`. It returns `1` if that is allowed in this arrangement, `0` if not.

<sup>25</sup> <http://www.nmt.edu/tcc/help/pubs/python22/>

For example, if the hierarchy contains class, order, family, optional subfamily, genus, and species, then a family can have a genus child (since the subfamily rank is optional), but an order cannot have any child other than a family.

**.formRank()**

If this hierarchy includes a form rank, this method returns it as a `Rank` object; otherwise it returns `None`.

**.genusRank()**

If this hierarchy includes a genus rank, this method returns it as a `Rank` object. If the genus rank is missing, returns `None`.

**.lookupRankCode(c)**

Used to find the rank corresponding to a rank code `c`. Returns that rank as a `Rank` object; see Section 9.3, “Class Rank: One taxonomic rank” (p. 22).

**.speciesRank()**

Returns this hierarchy's species rank as a `Rank` object. If for some bizarre reason there is no species rank, it returns otherwise it returns `None`.

**.subgenusRank()**

If this hierarchy includes a subgenus rank, this method returns it as a `Rank` object; otherwise it returns `None`.

**.\_\_getitem\_\_(self, key)**

Called when a `Hier` object `H` is indexed using the construct “`H[i]`”. If the  $0 \leq i < \text{len}(H)$ , the method returns the  $n$ th rank, counting the root rank as 0, the next deeper rank as 1, and so on.

For example, if `h` is a `Hier` object whose first three ranks are class, order, and family, `h[2]` returns the family rank.

**.\_\_iter\_\_(self)**

This is the iterator function for a `Hier` object. It iterates over the contained ranks from the root rank to the deepest. This method is called, for example, when you use a `Hier` object in a `for` statement.

For example, if `h` is a `Hier` object, this loop will print the names of all the ranks:

```
for rank in h:
    print rank.name
```

**.\_\_len\_\_(self)**

This method is called when the “`len()`” function is invoked on a `Hier` object. It returns the number of ranks in that object, as an integer.

### 9.3. Class Rank: One taxonomic rank

An object of this class represents one of the taxonomic ranks or levels such as order, family, genus, and so on.

Attributes of a `Rank` object include:

**.code**

The short code for this rank; see Section 5.1, “The ranks file” (p. 7) for a discussion of rank codes.

**.depth**

Depth of the rank within the hierarchy: 0 for the root rank, 1 for the next deeper rank, and so on. This value is also the index of the rank within the containing `Hier` object. For example, if you have a `Hier` object `h`, `h[n]` will have a `.depth` attribute of `n`.

**.hier**

The containing `Hier` object.

**.isOptional**

True if this rank is not required. For example, some bird families are divided into subfamilies but some are not, so the subfamily `rank` object will have this attribute set to 1.

**.keyLen**

The number of digits in the taxonomic key for this rank.

**.name**

The name of the rank, e.g., "Genus".

## 9.4. The `Taxon` class: One node in the classification tree

Each instance of a `Taxon` object represents one taxon. These objects make up the tree representing the taxonomic arrangement.

See Section 9.1, "Class `Txny`: the complete system" (p. 20) for various ways to obtain `Taxon` objects corresponding to a given scientific name, bird code, or taxonomic key number.

Attributes of a `Taxon` object:

**.txny**

A pointer back to the `Txny` object in which this taxon is located.

**.abbr**

The standard bird code, if this taxon has one. Higher ranks such as families and orders may not have a standard code.

**.eng**

The English name of the taxon, as a string.

**.engComma**

The English name of the taxon, as a string, but with multi-word names inverted as "last, first", e.g., "Owl, Great Gray" instead of "Great Gray Owl".

**.parent**

For the root taxon, this attribute is `None`. For the other taxa in the tree, it points to the taxon that has this taxon as a child.

**.rank**

The rank of this taxon, as a `Rank` object.

**.sci**

The scientific name of the taxon, as a string.

**.status**

The status code for this taxon. For the values of status codes, see Section 5.2.2, "Species records in the `.std` file" (p. 10).

**.tex**

Normally, this attribute has a value of `None`. However, if the English name of this taxon has markup (such as italics or non-ASCII characters), this attribute may contain the English name using `TEX` markup conventions.

**.txKey**

The taxonomic key number for this taxon.

These methods are available on `Taxon` objects:

**.contains(t2)**

If `t2` is another `Taxon` object in the same `Txny`, this predicate tests whether `self` contains `t2`.

**.nearestAncestor(t2)**

If `t2` is another `Taxon` object in the same `Txny`, this method returns the nearest ancestor of `self` and `t2` as a `Taxon` object.

**.childContaining(desc)**

If `desc` is a `Taxon` descended from `self`, this method returns the child of `self` that contains `desc`.

If `desc` is not a descendant of `self`, the method raises `ValueError`.

**.\_\_cmp\_\_(self, other)**

This method implements the `cmp()` function for comparing two `Taxon` objects. The comparison is in phylogenetic order, that is, the taxon with the lower `.txKey` attribute will compare lower.

**.\_\_getitem\_\_(self, n)**

If  $0 \leq n < \text{len}(\text{self})$ , returns the  $n$ th child of `self`; otherwise it raises `KeyError`. This method is called when a `Taxon` object `t` is indexed as `t[n]`.

**.\_\_iter\_\_(self)**

Defines the `iter()` function on a `Taxon` element. In constructs such as “for child in taxon”, this function returns an iterator that visits each child in order, 0, 1, ...

**.\_\_len\_\_(self)**

Returns the number of child taxa.

## 10. The `abbr.py` module

---

Assorted machinery having to do with just the bird code system is relegated to a separate module, `abbr.py`.

The module contains an assortment of manifest constants and functions, and one class. The constants and functions are:

**ABBR\_L**

Maximum length of a bird code, 6 in the CBC system.

**BLANK\_ABBR**

A string containing `ABBR_L` spaces.

**RE\_ABBR**

A regular expression (using Python's standard `re` regular expression module) that describes a valid bird code.

**REL\_SIMPLE**

The relationship code for simple (non-compound) forms, one space.

**REL\_HYBRID**

The single-character relationship code denoting hybrids, “^”.

**REL\_PAIR**

The single-character relationship code denoting a species pair, “|”.

**abbreviate(eng)**

Abbreviates an English name according to the rules of the system. Takes a string containing a name either in the usual word order (e.g., “Aztec Thrush”) or in “last, first” order (e.g., “Amakihi, Molokai”).

### **engComma (eng)**

Given an English name in the customary order, such as “American Robin”, returns it in the inverted form, e.g., “Robin, American”.

### **engDeComma (eng)**

Given an English name in the inverted form, such as “Robin, American”, returns the customary form, e.g., “American Robin”.

## 10.1. class BirdId

Representation of Christmas Bird Count data is complicated considerably by the use of what we call *compound forms*: species pairs (e.g., “Hammond's/Dusky Flycatcher”) and hybrids (e.g., “Baltimore Oriole×Bullock's Oriole”). Also supported is a trailing “?” to indicate that the identification is only a guess.

Here is the interface to the `BirdId` class, which represents simple and compound forms and an optional question mark.

### **BirdId ( txny, abbr, rel=None, abbr2=None, q=None )**

Because `BirdId` objects connect bird codes to a firm taxonomic foundation, you must pass a `Txny` object as the first argument to the constructor.

The second argument is a bird code. It can be in either upper or lower case, and either variable-length or right-padded with spaces. It will be stored in normalized form: uppercased and right-padded with spaces to length `ABBR_L`.

For single bird identities, omit the remaining arguments. For hybrids, pass `rel=REL_HYBRID` and the second bird code in the `abbr2` argument.

The `q` argument should be the string “?” if the ID is questionable. The default value is `None`, meaning that the ID is not in question.

Here's an example. Suppose `txny` is your `Txny` object. This code snippet sets `b1` to a `BirdId` object representing `Ou` (a Hawaiian endemic), and `b2` to a `BirdId` object representing `Indigo × Lazuli Bunting`:

```
b1 = BirdId ( txny, "ou" )
b2 = BirdId ( txny, "lazbun", REL_HYBRID, "indbun" )
```

This constructor will raise a `KeyError` exception if any of the abbreviations are undefined in `txny`.

#### **.txny**

The `.txny` attribute of a `BirdId` object is the `Txny` object passed to the constructor (read-only).

#### **.abbr**

The first or only bird code, normalized. A normalized code is uppercased, and right-padded with spaces if necessary to length `ABBR_L`.

#### **.rel**

For single forms, this attribute is `None`. It is set to `REL_HYBRID` for hybrids, `REL_PAIR` for species pairs.

#### **.abbr2**

For compound forms, this attribute holds the second bird code, normalized.

We stipulate that for any `BirdId` instance `B`, `B.abbr < B.abbr2`. This means that if you're looking for a specific hybrid or pair, you don't have to look in two different places. So we swap the `.abbr` and `.abbr2` values if necessary to make this true. For example, in the object returned by

"b2 = BirdId ( txny, "lazbun", REL\_HYBRID, "indbun" )" b2.abbr would be "indbun", and "lazbun" would be stored in b2.abbr2.

#### **.q**

Has the value "" (the empty string) if the ID is not in question; "?" if there is a question about the ID; or "-" if the ID is correct but the form is not countable under American Birding Association rules.

#### **.taxon**

This attribute will contain a `Taxon` object representing the smallest taxon that contains this identity. For a single form, this will be taxonomic key of the taxon containing the form. For hybrids and species pairs, it will be taken from the smallest taxon that is an ancestor of both forms.

#### **.fullAbbr**

Contains a string made from self's `.abbr` attribute, with the `.rel` and `.abbr2` attributes concatenated only for compound forms. Short codes are blank-stripped.

#### **.engComma()**

Returns the English name of `self` in inverted order, that is, "last, first". Examples: "robin, American"; "mallard x teal, blue-winged"; "ibis, glossy?".

#### **.\_\_str\_\_(self)**

This method is called when a `BirdId` object is converted to a string, implicitly or by explicit use of the `str()` function. It returns the English name as a string. Examples of its return values: "Nihoa Finch"; "Blue-winged Teal x Cinnamon Teal"; "Dusky Flycatcher / Hammond's Flycatcher".

### **BirdId.scan ( txny, scan )**

This method works with the `Scan` object, from the author's personal Python library, to process raw bird codes while scanning an input file. For more information on the `Scan` object, see the author's library reference<sup>26</sup>.

This is a static method, a relatively new feature of Python. For more information on Python static methods, see the *Python 2.2 quick reference*<sup>27</sup>.

The `txny` argument is a `Txny` object providing the taxonomy system in which the codes are to be interpreted. The `scan` argument is a `Scan` object used to scan the input stream containing the bird codes.

This method looks for a bird code, optionally followed by a relationship code and a second bird code (which we call a *compound code*). Examples: "vireo", for "vireo sp."; "mallar^amewig", Mallard x American Wigeon; and "dowwoo|haiwoo", Downy or Hairy Woodpecker.

If the `scan` object points at a valid simple or compound code, the `scan` object is advanced past that code, and the method returns a new `BirdId` object representing the code. If the `scan` object doesn't start with a valid code, an error message is sent to the `scan` object's error log, and a `ValueError` exception is raised.

This method will recognize a trailing "?" if present.

The method raises `KeyError` if any bird codes are undefined.

### **BirdId.scanFlat ( txny, scan )**

This is another static method like `BirdId.scan()`, but it expects to see its input in flat file format. Specifically, the `scan` object should start with three fixed fields. The first field has length `ABBR_L` and contains the first or only bird code, left-aligned and right-padded with spaces. The second field

<sup>26</sup> <http://infohost.nmt.edu/~shipman/soft/clean/lib.html>

<sup>27</sup> <http://www.nmt.edu/tcc/help/pubs/python22/recent-features.html#static-methods>

is a single character and contains the relationship code: normally blank, but it may contain REL\_HYBRID or REL\_PAIR for compound codes. The third field has length ABBR\_L and contains the second bird code when the relationship code is nonblank. The third field must be blank when the relationship code is blank.

This method does not support the questionable ID flag. If any codes are undefined, it raises `KeyError`.

**BirdId.parse ( txny, s )**

This static method is also like `BirdId.scan()`, but is used when the input is in an ordinary string instead of a `Scan` object.

This method supports a trailing "?" for questionable IDs. It will raise `KeyError` for undefined codes.

