

Bird taxonomy system: internal maintenance specification

Zoological
Data Processing

ℒxml version

John W. Shipman

2011-06-06 13:39

Abstract

Describes the implementation of a system for representing a classification of a set of birds, using the Python programming language and XML (Extensible Markup Language).

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

Table of Contents

1. Introduction	2
2. The <code>txny.py</code> module: prologue	3
3. Some XML helper routines	4
3.1. <code>getFirstChild()</code> : Get the first child of a given name	4
3.2. <code>textContent()</code> : Retrieve the text of a node	4
3.3. <code>getChildContent()</code> : Get text from a child node	4
4. class <code>Txny</code>	5
4.1. <code>Txny.genAbbrs()</code> : Generate all valid bird codes	7
4.2. <code>Txny.genTxKeys()</code> : Generate all phylogenetic key numbers	7
4.3. <code>Txny.lookupAbbr()</code>	7
4.4. <code>Txny.lookupCollision()</code>	7
4.5. <code>Txny.lookupSci()</code>	8
4.6. <code>Txny.lookupTxKey()</code>	8
4.7. <code>Txny.abbrToEng()</code>	8
4.8. <code>Txny.abbrToTex()</code>	8
4.9. <code>Txny.__init__()</code>	9
4.10. <code>Txny.__readTaxonomy()</code>	10
4.11. <code>txny.__buildMaps()</code> : Index by key and scientific name	11
4.12. <code>Txny.__addTaxonMaps()</code> : Add a taxon to the internal dictionaries	11
4.13. <code>Txny.__readAbbrs()</code>	12
4.14. <code>Txny.__readCollisions()</code>	13
4.15. <code>Txny.__addCollision()</code> : Add one collision	14
5. class <code>Hier</code>	14
5.1. <code>Hier.__init__()</code>	16
5.2. <code>Hier.canParentHaveChild()</code>	16

¹ <http://www.nmt.edu/~shipman/xnomo/ims2/>

² <http://www.nmt.edu/~shipman/xnomo/ims2/xnomoims.pdf>

5.3. Hier.formRank(), Hier.genusRank(), Hier.speciesRank(), and Hier.sub-genusRank()	17
5.4. Hier.lookupRankCode()	17
5.5. Hier.__getitem__()	18
5.6. Hier.__iter__()	18
5.7. Hier.__len__()	18
6. class Rank: One taxonomic rank	18
6.1. Rank.__init__()	18
6.2. Rank.__str__()	19
7. class Taxon	20
7.1. Taxon.contains()	21
7.2. Taxon.nearestAncestor()	22
7.3. Taxon.childContaining(): Find child containing a descendant	22
7.4. Taxon.__cmp__(): Compare taxa in phylogenetic order	23
7.5. Taxon.__getitem__()	24
7.6. Taxon.__iter__()	24
7.7. Taxon.__len__()	24
7.8. Taxon.__str__()	24
7.9. Taxon.__init__(): The constructor	25
8. The abbr.py module	26
8.1. Prologue	26
8.2. abbreviate()	27
8.3. engComma(): Invert an English name	29
8.4. engDeComma(): Normalize an English name	30
8.5. class BirdId	30
8.6. BirdId.__init__()	32
8.7. BirdId.normalize()	33
8.8. birdId.engComma(): Inverted English name	34
8.9. BirdId.commafy(): Invert word order (static method)	34
8.10. BirdId.__str__()	35
8.11. BirdId.scan()	35
8.12. BirdId.scanFlat()	37
8.13. BirdId.scanAbbr()	38
8.14. BirdId.scanAbbrFlat()	39
8.15. BirdId.parse(): Parse a code from a string	40

1. Introduction

This document describes the internals of the programs associated with *A system for representing bird taxonomy*³. Refer to that document for downloadable online files.

Here, in literate style, is the source code for the Python objects in the `txny.py` module.

In addition to its utility in bird records management, this module is also an example of several useful techniques:

- Literate programming, the writing of code and documentation as a single process with a single source file. See *A source extractor for lightweight literate programming*⁴.

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

⁴ <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/>

- Cleanroom (zero-defect) programming. See *Toward Zero-defect Programming* by Allan M. Stavelly (Addison-Wesley, 1999, ISBN 0-201-38595-3) for an overview of this methodology, and see also the author's Cleanroom pages⁵ for details of the method as he practices it.
- XML processing using the Python lxml package; see *Python XML processing with lxml*⁶.

2. The txny.py module: prologue

The txny.py module starts with a module documentation string in the standard Python format.

```
txny.py
```

```
"""txny.py:  Taxonomy objects for bird records work.

For full documentation, see:
    http://www.nmt.edu/~shipman/xnomo/
"""
```

Next are imported modules. The lxml.etree package handles XML parsing; see *Python XML processing with lxml*.⁷

```
txny.py
```

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

import lxml.etree as et
```

File rnc_txny.py contains symbolic constants for all the XML element and attribute names used in the schema. It is created automatically from the schema: see *pyrang: A single-sourcing tool for Python-XML applications*.⁸

```
txny.py
```

```
import rnc_txny as rnc
```

Then a section of manifest constants global to the module.

```
txny.py
```

```
#=====
# Manifest constants
#  DEFAULT_FILE_NAME:  Where to look for input by default.
#  ABBR_L:             Bird code length
#  GENUS_RANK_CODE:   Rank code for genus
#  SUBGENUS_RANK_CODE: Rank code for subgenus
#  SPECIES_RANK_CODE: Rank code for species
#  FORM_RANK_CODE:    Rank code for form
#-----

DEFAULT_FILE_NAME = "aou.xml"
ABBR_L            = 6
GENUS_RANK_CODE  = "g"
SUBGENUS_RANK_CODE = "-g"
```

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

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

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

⁸ <http://www.nmt.edu/tcc/help/lang/python/examples/pyrang>

```
SPECIES_RANK_CODE = "s"
FORM_RANK_CODE    = "x"
```

3. Some XML helper routines

Here are some routines to perform common XML extraction tasks.

3.1. `getFirstChild()`: Get the first child of a given name

Because we validate the input file against the schema, we can get away without a lot of error checking. This routine is used to retrieve a child node of a given element name. It will fail if there is no child node by that name.

txny.py

```
# - - -   g e t F i r s t C h i l d

def getFirstChild ( node, childName ):
    """Get the first child of a given name.

    [ (node is an et.Element) and
      (childName is a string) ->
        if node has any child elements named (childName) ->
          return the first such child element
        else -> raise IndexError ]
    """
    return node.xpath ( childName ) [0]
```

3.2. `textContent()`: Retrieve the text of a node

This function returns the concatenation of all direct text children of a given node. The result will be an empty string if there are no text children. If there are any child nodes with text, their text will not be included.

Because of the strange way `lxml` handles text, we'll use an XPath expression to do the work.

txny.py

```
# - - -   t e x t C o n t e n t

def textContent ( node ):
    """Return a node's text content.

    [ node is an et.Element ->
      return the concatenation of all direct text children of
      node ]
    """
    return "".join ( node.xpath ( 'text()' ) )
```

3.3. `getChildContent()`: Get text from a child node

This function returns the text content of a node's child.

```
# - - -   g e t C h i l d C o n t e n t

def getChildContent ( node, childName ):
    """Get the text from a child node.

    [ (node is an et.Element) and
      (childName is a string) ->
        if node has any children named childName ->
          return the concatenation of the first such child
          node's direct text content
        else -> return "" ]
    """
    childList = node.xpath ( childName )
    if len(childList) == 0:
        return ""
    else:
        return textContent ( childList[0] )
```

4. class Txny

We start off with class Txny. Here is its external interface:

```
# - - - - -   c l a s s   T x n y   - - - - -

class Txny:
    """Object to represent the entire taxonomy and code system.

    Exports:
    Txny ( dataFile=None ):
        [ dataFile is a string, defaulting to DEFAULT_FILE_NAME ->
          if dataFile names a readable XML file that
          validates against txny.rnc ->
            return a new Txny object representing that file
          else -> raise IOError ]
    .genAbbrs():
        [ generate all valid bird codes in self, in ascending
          order ]
    .genTxKeys():
        [ generate all the taxonomy keys in self, in ascending
          order ]
    .lookupAbbr ( abbr ):
        [ abbr is a string ->
          if self contains a Taxon whose .abbr attribute
          matches abbr, case-insensitive and blank-padded ->
            return that Taxon
          else -> raise KeyError ]
    .lookupCollision ( abbr ):
        [ abbr is a string ->
          if self contains a collision code matching abbr,
```

```

        case-insensitive ->
            return a list of the valid alternative codes
        else -> raise KeyError ]
.lookupSci ( sci ):
    [ sci is a string ->
        if self contains a taxon whose scientific name matches
        sci ->
            return that taxon as a Taxon object
        else -> raise KeyError ]
.lookupTxKey ( txKey ):
    [ txKey is a string ->
        if self contains a Taxon whose .txKey attribute
        matches txKey ->
            return that Taxon
        else -> raise KeyError ]
.abbrToEng ( abbr ):
    [ abbr is a string ->
        if abbr is a bird code in self ->
            return the corresponding English name
        else -> raise KeyError ]
.abbrToTex ( abbr ):
    [ abbr is a string ->
        if abbr is a bird code in self ->
            return the corresponding TeX name
        else -> raise KeyError ]

```

The various lookup functions require that we have internal dictionaries to map the various names, keys and codes to `Taxon` objects.

txny.py

State/Invariants:

```

.__abbrMap:
    [ a dictionary whose keys are the bird codes in self,
      uppercased, and their corresponding values are the related
      Taxon objects ]
.__collMap:
    [ a dictionary whose keys are the collision codes in self,
      uppercased, and their corresponding values are lists
      of the valid substitute codes ]
.__sciMap:
    [ a dictionary whose values are the Taxon objects in self
      and the corresponding keys are their .sci attributes ]
.__txKeyMap:
    [ a dictionary whose values are the Taxon objects in self
      and the corresponding keys are their .txKey attributes ]
.__abbrEng:
    [ a dictionary whose values are the English names from which
      the bird codes in self were derived, and the corresponding
      keys are those codes ]
.__abbrTex:
    [ a dictionary whose values are the English names from which
      the bird codes in self were derived, marked up with TeX

```

```
""" conventions, and the corresponding keys are those codes ]
```

4.1. Txny.genAbbrs(): Generate all valid bird codes

Generates the keys from the `.__abbrMap` dictionary, in ascending order. Note that the keys are already uppercased.

txny.py

```
# - - - T x n y . g e n A b b r s - - -  
  
def genAbbrs ( self ):  
    """Generate all valid bird codes in self.  
    """  
    keyList = self.__abbrMap.keys()  
    keyList.sort()  
    for key in keyList:  
        yield key  
    raise StopIteration
```

4.2. Txny.genTxKeys(): Generate all phylogenetic key numbers

Generates all the keys in the `.__txKeyMap` dictionary, in ascending order.

txny.py

```
# - - - T x n y . g e n T x K e y s - - -  
  
def genTxKeys ( self ):  
    """Generate all taxonomic keys in self.  
    """  
    keyList = self.__txKeyMap.keys()  
    keyList.sort()  
    for key in keyList:  
        yield key  
    raise StopIteration
```

4.3. Txny.lookupAbbr()

Retrieves the Taxon for a given bird code, if any.

txny.py

```
# - - - T x n y . l o o k u p A b b r - - -  
  
def lookupAbbr ( self, abbr ):  
    """Look up a taxon by bird code.  
    """  
    return self.__abbrMap [ abbr.upper().ljust(ABBR_L) ]
```

4.4. Txny.lookupCollision()

Checks a code to see if it is a collision form. If it is, returns a list of the valid alternatives.

```
# - - - T x n y . l o o k u p C o l l i s i o n - - -

def lookupCollision ( self, abbr ):
    """Is abbr a collision code?
    """
    return self.__collMap [ abbr.upper().ljust(ABBR_L) ]
```

4.5. Txny.lookupSci()

Looks up a taxon by its scientific name.

```
# - - - T x n y . l o o k u p S c i - - -

def lookupSci ( self, sci ):
    """Look up a taxon by scientific name.
    """
    return self.__sciMap [ sci ]
```

4.6. Txny.lookupTxKey()

Retrieves a TAXON by its numeric key.

```
# - - - T x n y . l o o k u p T x K e y - - -

def lookupTxKey ( self, txKey ):
    """Look up a taxon by phylogenetic key number.
    """
    return self.__txKeyMap [ txKey ]
```

4.7. Txny.abbrToEng()

This method uppercases its argument and looks it up in the self.__abbrEng dictionary.

```
# - - - T x n y . a b b r T o E n g - - -

def abbrToEng ( self, abbr ):
    """Lookup the English name corresponding to abbr.
    """
    return self.__abbrEng [ abbr.upper().ljust(ABBR_L) ]
```

4.8. Txny.abbrToTex()

This method uppercases and blank-fills its argument and looks it up in the self.__abbrTex dictionary.

```
# - - - T x n y . a b b r T o T e x - - -

def abbrToTex ( self, abbr ):
```

```

"""Lookup the TeX name corresponding to abbr.
"""
return self.__abbrTex [ abbr.upper().ljust(ABBR_L) ]

```

4.9. Txny.__init__()

Here is the class constructor, which takes as its argument the name of the XML data file.

txny.py

```

# - - - T x n y . _ _ i n i t _ _ - - -

def __init__ ( self, dataFile=None ):
    """Constructor for the Txny object.
    """

```

First we figure out the actual file name we're reading:

txny.py

```

#-- 1 --
if dataFile is None:
    fileName = DEFAULT_FILE_NAME
else:
    fileName = dataFile

```

The next step is to read the entire XML file. This will succeed if the file is readable and well-formed, but it will raise an exception otherwise.

txny.py

```

#-- 2 --
# [ if fileName names a readable, well-formed XML file ->
#     doc := a Document node representing that file
#     root := an Element node representing the file's
#           root element
# else -> raise IOError ]
try:
    doc = et.parse ( fileName )
    root = doc.getroot()
except Exception, detail:
    raise IOError, ( "Error reading the taxonomy file "
                    "'%s': %s" %
                    (fileName, detail) )

```

First we process the rankSet element, which returns a Hier object:

txny.py

```

#-- 3 --
# [ self.hier := a new Hier object made from root's
#              RANK_SET_N subtree ]
rankSetNode = root.xpath ( rnc.RANK_SET_N )[0]
self.hier = Hier ( rankSetNode )

```

Next we process the taxonomic tree:

txny.py

```

#-- 4 --
# [ self.root := a Taxon object made from root's TAXONOMY_N
#              subtree, with children representing that subtree

```

```

# self.__txKeyMap := as invariant, from that subtree
# self.__sciMap   := as invariant, from that subtree ]
taxonomyNode = root.xpath ( rnc.TAXONOMY_N )[0]
self.__readTaxonomy ( taxonomyNode )

```

Now, read the set of valid bird codes, then the set of collisions:

txny.py

```

#-- 5 --
# [ self.__abbrMap := as invariant, from root's ABBR_SET_N
#       subtree
#   self.__abbrEng := as invariant, from that subtree
#   self.__abbrTex := as invariant, from that subtree ]
abbrSetNode = root.xpath ( rnc.ABBR_SET_N )[0]
self.__readAbbrs ( abbrSetNode )

#-- 6 --
# [ self.__collMap := as invariant, from root's
#       COLLISION_SET_N subtree ]
collSetNode = root.xpath ( rnc.COLLISION_SET_N )[0]
self.__readCollisions ( collSetNode )

```

4.10. Txny.__readTaxonomy()

This method reads the root taxonomy node and recursively adds its descendants.

txny.py

```

# - - - T x n y . _ _ r e a d T a x o n o m y - - -

def __readTaxonomy ( self, taxonomyNode ):
    """Build the taxonomy subtree.

    [ taxonomyNode is a TAXONOMY_N Element node ->
      self.root := a Taxon object made from root's TAXONOMY_N
                  subtree, with children representing that subtree
      self.__txKeyMap := as invariant, from that subtree
      self.__sciMap   := as invariant, from that subtree ]
    """

```

First we find the `taxon` node that roots the classification tree:

txny.py

```

#-- 1 --
# [ rootTaxonNode := the TAXON_N node child of taxonomyNode ]
rootTaxonNode = taxonomyNode.xpath ( rnc.TAXON_N )[0]

```

Then we recursively transform that node and its descendants into a tree of `Taxon` objects, storing the new root object in `self.root`:

txny.py

```

#-- 2 --
# [ self.root := a new Taxon object representing rootTaxonNode
#       and having descendants made from the descendants of
#       rootTaxonNode ]
self.root = Taxon ( self, None, rootTaxonNode )

```

We still need to add entries to our internal maps so we can find taxa by taxonomic key or by scientific name:

txny.py

```
#-- 3 --
# [ self.__txKeyMap := as invariant, from self.root and its
#     descendants
#   self.__sciMap   := as invariant, from self.root and its
#     descendants ]
self.__buildMaps()
```

4.11. txny.__buildMaps(): Index by key and scientific name

This method builds the internal `.__txKeyMap` and `.__sciMap` dictionaries.

txny.py

```
# - - -   T x n y . _ _ b u i l d M a p s   - - -
def __buildMaps ( self ):
    """Build the key and scientific name maps.

    [ self.__txKeyMap := as invariant, from self.root and its
      descendants
      self.__sciMap   := as invariant, from self.root and its
      descendants ]
    """

    #-- 1 --
    self.__txKeyMap = {}
    self.__sciMap   = {}
```

Recursion takes care of visiting every Taxon node starting at `self.root`.

txny.py

```
#-- 2 --
# [ self.__txKeyMap += entries whose values are the Taxon
#     objects rooted in self.root and whose keys are the
#     .txKey attributes of those Taxon objects
#   self.__sciMap   += entries whose values are the Taxon
#     objects rooted in self.root and whose keys are the
#     .sci attributes of those Taxon objects ]
self.__addTaxonMaps ( self.root )
```

4.12. Txny.__addTaxonMaps(): Add a taxon to the internal dictionaries

txny.py

```
# - - -   T x n y . _ _ a d d T a x o n M a p s   - - -
def __addTaxonMaps ( self, taxon ):
    """Add a taxon and its descendants to the internal dictionaries.

    [ taxon is a Taxon object ->
      self.__txKeyMap += entries whose values are the Taxon
      objects rooted at taxon and whose keys are the
```

```

        .txKey attributes of those Taxon objects
        self.__sciMap += entries whose values are the Taxon
        objects rooted at taxon and whose keys are the
        .sci attributes of those Taxon objects ]
    """

```

This method is called recursively to visit every Taxon object in the tree. First we take care of the basis case, the Taxon object where we start:

txny.py

```

#-- 1 --
# [ self.__txKeyMap += an entry whose value is taxon and
#     whose key is taxon.txKey
#   self.__sciMap   += an entry whose value is taxon and
#     whose key is taxon.sci ]
self.__txKeyMap[taxon.txKey] = taxon
self.__sciMap[taxon.sci]     = taxon

```

Then recursively evaluate the subtrees of this taxon:

txny.py

```

#-- 2 --
# [ self.__txKeyMap += entries whose values are the
#     descendants of taxon and whose keys are the .txKey
#     attributes of those descendants
#   self.__sciMap   += entries whose values are the
#     descendants of taxon and whose keys are the .sci
#     attributes of those descendants ]
for child in taxon:
    self.__addTaxonMaps ( child )

```

4.13. Txny.__readAbbrs()

This method handles building the .__abbrMap dictionary from the ABBR_SET_N node.

txny.py

```

# - - - T x n y . _ _ r e a d A b b r s - - -

def __readAbbrs ( self, abbrSetNode ) :
    """Read the ABBR_SET_N subtree and build self.__abbrMap.

    [ abbrSetNode is a ABBR_SET_N node ->
      self.__abbrMap := a dictionary whose keys are the
                       valid bird codes in self, with the corresponding
                       values the Taxon object for each code
      self.__abbrEng := as invariant, from that subtree
      self.__abbrTex := as invariant, from that subtree ]
    """

    #-- 1 --
    # [ self.__abbrMap := a new, empty dictionary
    #   self.__abbrEng := a new, empty dictionary
    #   self.__abbrTex := a new, empty dictionary
    #   abbrNodeList   := list of the ABBR_N children of abbrSetNode
    ]

```

```

self.__abbrMap = {}
self.__abbrEng = {}
self.__abbrTex = {}
abbrNodeList = abbrSetNode.xpath ( rnc.ABBR_N )

#-- 2 --
# [
for abbrNode in abbrNodeList:
    #-- 2 body --
    # [ abbrNode is a ABBR_N Element whose SCI_A attribute
    #   is a scientific name defined in self.__sciMap ->
    #   self.__abbrMap += an entry mapping the uppercased
    #   CODE_A attribute of abbrNode |-> the Taxon
    #   in self with that scientific name
    #   self.__abbrEng += an entry mapping the uppercased
    #   CODE_A attribute of abbrNode |-> the text content
    #   of abbrNode
    #   self.__abbrTeX += an entry mapping the uppercased
    #   CODE_A attribute of abbrNode |-> the content of
    #   of the TEX_NAME_N child of abbrNode, if any,
    #   defaulting to the text content of abbrNode ]
    abbr = abbrNode.attrib[rnc.CODE_A]
    upAbbr = abbr.upper().ljust(ABBR_L)
    sci = abbrNode.attrib[rnc.SCI_A]
    eng = textContent ( abbrNode ).strip()
    tex = getChildContent ( abbrNode, rnc.TEX_NAME_N )
    if not tex:
        tex = eng
    taxon = self.__sciMap[sci]
    self.__abbrMap[upAbbr] = taxon
    self.__abbrEng[upAbbr] = eng
    self.__abbrTex[upAbbr] = tex

```

4.14. Txny.__readCollisions()

This method builds the `.__collMap` dictionary using the `COLLISION_SET_N` node.

txny.py

```

# - - -   T x n y . _ _ r e a d C o l l i s i o n s   - - -

def __readCollisions ( self, collSetNode ):
    """Read the COLLISION_SET_N node and build self.__collMap.
    """

    #-- 1 --
    self.__collMap = {}

    #-- 2 --
    # [ self.__collMap += entries mapping BAD_ABBR_A attributes
    #   from children of collSetNode |-> lists of the codes
    #   from the content of GOOD_ABBR_N element children of
    #   those children ]
    for collNode in collSetNode.xpath ( rnc.COLLISION_N ):

```

```

    #-- 2 body --
    # [ collNode is a COLLISION_N Element ->
    #     self.__collMap += an entry mapping the BAD_ABBR_A
    #         attribute of collNode |-> a list of the codes
    #         from the content of GOOD_ABBR_N element children
    #         of collNode ]
    self.__addCollision ( collNode )

```

4.15. Txny.__addCollision(): Add one collision

Converts a COLLISION_N node to an entry in self.__collMap.

txny.py

```

# - - -   T x n y . _ _ a d d C o l l i s i o n   - - -

def __addCollision ( self, collNode ):
    """Convert a COLLISION_N node to a __collMap entry.

    [ collNode is a COLLISION_N Element ->
      self.__collMap += an entry mapping the BAD_ABBR_A
        attribute of collNode |-> a list of the codes
        from the content of GOOD_ABBR_N element children
        of collNode ]

    """

    #-- 1 --
    # [ badAbbr   := collNode's BAD_ABBR_A attribute
    #   goodList  := GOOD_ABBR_N children of collNode ]
    badAbbr = collNode.attrib[rnc.BAD_ABBR_A]
    goodNodeList = collNode.xpath ( rnc.GOOD_ABBR_N )

    #-- 2 --
    # [ goodAbbrs := a list of the contents of goodNodeList ]
    goodAbbrs = [ textContent(goodNode).strip()
                  for goodNode in goodNodeList ]

    #-- 3 --
    self.__collMap [ badAbbr.upper() ] = goodAbbrs

```

5. class Hier

The Hier object is a container for the set of taxonomic ranks used in the classification. Its constructor is not designed to be called directly; that constructor is used only in the initialization of the Txny object.

txny.py

```

# - - - - -   c l a s s   H i e r   - - - - -

class Hier:
    """Represents the set of ranks used in the classification.

    Exports:

```

```

Hier ( rankSetNode ):
    [ rankSetNode is a RANK_SET_N Element ->
      return a new Hier object made from rankSetNode ]
.txKeyLen:
    [ number of characters in a taxonomic key with this
      hierarchy ]
.canParentHaveChild ( p, c ):
    [ p and c are Rank objects ->
      if there exists a non-optional rank m in self.__rankList
      such that p.depth < m.depth < c.depth ->
      return 0
      else -> return 1 ]
.genusRank():
    [ if self has a genus rank ->
      return that rank as a Rank object
      else -> return None ]
.subgenusRank():
    [ if self has a subgenus rank ->
      return that rank as a Rank object
      else -> return None ]
.speciesRank():
    [ if self has a species rank ->
      return that rank as a Rank object
      else -> return None ]
.formRank():
    [ if self has a form rank ->
      return that rank as a Rank object
      else -> return None ]
.lookupRankCode ( rankCode ):
    [ rankCode is a string ->
      if there is a rank in self whose .code equals rankCode ->
      return the corresponding Rank object
      else -> raise KeyError ]
.__getitem__ ( self, n ):
    [ n is an integer ->
      if n is the rank depth of a rank in self ->
      return that rank
      else -> raise IndexError ]
.__iter__ ( self ):
    [ return an iterator that yields the ranks in self
      from highest to deepest ]
.__len__ ( self ):
    [ return the number of ranks in self ]

```

State/Invariants:

```

.__rankList:
    [ a list of the Rank objects in self from highest to deepest ]
.__rankCodeMap:
    [ a dictionary whose values are the Rank objects in self,
      and for each value the key is its .code attribute ]
.__currentRankIndex:
    [ index in self.__rankList when self is an iterator ]

```

"""

5.1. Hier.__init__()

This is the constructor for the Hier class. It takes as an argument a `Element` node representing the `rankSet` element.

txny.py

```
# - - -   H i e r . _ _ i n i t _ _   - - -

def __init__ ( self, rankSetNode ):
    """Constructor for the Hier object.

    [ rankSetNode is a RANK_SET_N Element ->
      return a new Hier object made from rankSetNode ]
    """

    #-- 1 --
    self.__rankList      = []
    self.__currentRankIndex = 0

    #-- 2 --
    # [ self.__rankList += Rank objects made from the
    #     RANK_N children of rankSetNode, in the same order ]
    for rankNode in rankSetNode.xpath ( rnc.RANK_N ):
        #-- 2 body --
        # [ rankNode is a RANK_N node ->
        #     self.__rankList += a Rank object made from rankNode ]

        self.__rankList.append ( Rank ( self, rankNode ) )

    #-- 3 --
    # [ self.txKeyLen := sum of all child rank .keyLen attributes
    #     self.__rankCodeMap := entries mapping R.code |-> R for
    #     R the set of all ranks in self ]
    self.txKeyLen      = 0
    self.__rankCodeMap = {}
    for rank in self.__rankList:
        self.txKeyLen += rank.keyLen
        self.__rankCodeMap[rank.code] = rank
```

5.2. Hier.canParentHaveChild()

The test for whether a parent of rank p can have a child of rank c can be rephrased like this: are there any ranks between p and c that are not optional?

txny.py

```
# - - -   H i e r . c a n P a r e n t H a v e C h i l d   - - -

def canParentHaveChild ( self, p, c ):
    """Can a taxon of rank p have an immediate child of rank c?
    """

    #-- 1 --
    for rankIndex in range ( p.depth+1, c.depth ):
        #-- 1 body --
```

```

# [ if self.__rankList[rankIndex] is not optional ->
#     return 0
#     else -> I ]
m = self.__rankList[rankIndex]
if not m.isOptional:
    return 0

#-- 2 --
return 1

```

5.3. Hier.formRank(), Hier.genusRank(), Hier.speciesRank(), and Hier.subgenusRank()

Each of these methods looks for a given rank and returns the Rank object with the appropriate code, if there is one. It returns None if there is no rank with that code.

txny.py

```

# - - -   H i e r . f o r m R a n k   - - -
# - - -   H i e r . g e n u s R a n k   - - -
# - - -   H i e r . s p e c i e s R a n k   - - -
# - - -   H i e r . s u b g e n u s R a n k   - - -

def formRank ( self ):
    """Return self's form rank, or None if there isn't one.
    """
    return self.__rankCodeMap.get ( FORM_RANK_CODE, None )

def genusRank ( self ):
    """Return self's genus rank, or None if there isn't one.
    """
    return self.__rankCodeMap.get ( GENUS_RANK_CODE, None )

def speciesRank ( self ):
    """Return self's species rank, or None if there isn't one.
    """
    return self.__rankCodeMap.get ( SPECIES_RANK_CODE, None )

def subgenusRank ( self ):
    """Return self's subgenus rank, or None if there isn't one.
    """
    return self.__rankCodeMap.get ( SUBGENUS_RANK_CODE, None )

```

5.4. Hier.lookupRankCode()

Returns the Rank object with a given rank code; raises KeyError if there is no such rank.

txny.py

```

# - - -   H i e r . l o o k u p R a n k C o d e   - - -

def lookupRankCode ( self, rankCode ):
    """Find a rank by its code.

```

```
"""
return self.__rankCodeMap [ rankCode ]
```

5.5. Hier.__getitem__()

This special method is called when a Hier object H is indexed as Hier[n].

txny.py

```
# - - - H i e r . _ _ g e t i t e m _ _ - - -
def __getitem__ ( self, n ):
    """Return the nth rank in self.
    """
    return self.__rankList[n]
```

5.6. Hier.__iter__()

Returns an iterator for self. In this case it steps through the ranks in self from root to deepest.

txny.py

```
# - - - H i e r . _ _ i t e r _ _ - - -
def __iter__ ( self ):
    """Return self as an iterator.
    """
    for rank in self.__rankList:
        yield rank
    raise StopIteration
```

5.7. Hier.__len__()

Returns the length of self, that is, the number of ranks.

txny.py

```
# - - - H i e r . _ _ l e n _ _ - - -
def __len__ ( self ):
    """Return the length of self (rank count)."""
    return len ( self.__rankList )
```

6. class Rank: One taxonomic rank

Each Rank object represents one taxonomic rank (e.g., order, family, species) used in the classification.

6.1. Rank.__init__()

Since users do not need to call this constructor directly, it functions only during the translation of XML nodes. Accordingly, the constructor takes two arguments: a pointer back to the containing Hier object, and the node describing the rank.

```
# - - - - - c l a s s   R a n k   - - - - -

class Rank:
    """Represents one taxonomic rank.

    Exports:
    Rank ( hier, rankNode ):
        [ (hier is the containing Hier object) and
          (rankNode is a RANK_N node) ->
          return a new Rank object in hier representing
          rankNode ]
    .code:      [ self's rank code ]
    .depth:     [ self's depth, 0 for the root ]
    .keyLen:    [ length of self's portion of the taxonomic key ]
    .name:      [ English name of the rank ]
    .isOptional:
        [ if this rank can be omitted -> 1
          else -> 0 ]
    """
    """
```

Here is the trivial constructor.

```
# - - -   R a n k .   _ _   i n i t   _ _   - - -

def __init__ ( self, hier, rankNode ):
    """Constructor for a Rank object.
    """

    #-- 1 --
    self.hier = hier

    #-- 2 --
    # [ self.code      := the CODE_A attribute of rankNode
    #   self.depth     := the DEPTH_A attribute of rankNode as int
    #   self.keyLen    := the DIGITS_A attribute of rankNode
    #   self.name      := the text content of rankNode
    #   self.isOptional := true iff rankNode has an
    #                     OPTIONAL_A attribute ]
    self.code      = rankNode.attrib[rnc.CODE_A]
    self.depth     = int ( rankNode.attrib[rnc.DEPH_A] )
    self.keyLen    = int ( rankNode.attrib[rnc.DIGITS_A] )
    self.name      = textContent ( rankNode )
    self.isOptional = int ( rankNode.attrib.get(
        rnc.OPTIONAL_A, 0) )
```

6.2. Rank.__str__()

This method displays a Rank object as a string. It is useful in debugging.

```
# - - -   R a n k . _ _ s t r _ _   - - -

def __str__ ( self ):
    """Return self as a string."""
    return ( "%2d %-2s %s (keyLen %d, optional %d)" %
            ( self.depth, self.code, self.name, self.keyLen,
              self.isOptional ) )
```

7. class Taxon

Each taxon in the tree is represented by one Taxon object.

```
# - - - - -   c l a s s   T a x o n   - - - - -

class Taxon:
    """Represents one taxonomic grouping of organisms.

    Exports:
    Taxon ( txny, parentTaxon, taxonNode ):
        [ (txny is the containing Txny object) and
          (parentTaxon is the parent Taxon object, or None for root) and

          (taxonNode is a TAXON_N Element) ->
            return a new Taxon object pointing back to txny, with
            parent=parentTaxon, representing taxonNode and its
            descendants ]
    .txny:          [ as passed to constructor, read-only ]
    .parent:        [ parentTaxon passed to constructor, read-only ]
    .abbr:
        [ if self has a standard abbreviation ->
          that abbreviation
          else -> None ]
    .eng:           [ self's English name ]
    .rank:          [ self's taxonomic rank as a Rank object ]
    .sci:           [ self's scientific name ]
    .tex:           [ self's scientific name in TeX format ]
    .txKey:
        [ self's taxonomic key, a string of digits that orders
          taxa in phylogenetic order ]
    .contains ( other ):
        [ other is a Taxon object in self.txny ->
          if self is an ancestor of other ->
            return 1
          else -> return 0 ]
    .nearestAncestor ( other ):
        [ other is a Taxon ->
          return the Taxon representing the nearest ancestor
          of self and other ]
    .__cmp__ ( self, other ):
```

```

    [ other is a Taxon ->
      if self precedes other ->
        return a negative number
      else if self is the same as other ->
        return 0
      else -> return a positive number ]
    .__getitem__( n ):
    [ n is an integer ->
      if 0 <= n < (number of self's children) ->
        return self's nth child, counting from 0
      else -> raise KeyError ]
    .__iter__( self ):
    [ return an iterator that generates self's children
      in taxonomic order ]
    .__len__( self ):
    [ return number of self's child taxa ]
    .__str__( self ):
    [ return a string representation of self ]

```

The internal state includes a list of self's child taxa, if any.

txny.py

```

State/Invariant:
    .__childList:
    [ a list of the direct child taxa of self, in phylogenetic
      order, if any, as Taxon objects ]
    """

```

The private `.__childList` attribute will hold pointers to self's children, if any.

7.1. Taxon.contains()

The argument is another `Taxon`. The method returns 1 if the other taxon is contained in `self`, 0 otherwise.

The logic is straightforward: if the nearest common ancestor of `self` and the other taxon is `self`, then `self` contains the other taxon.

txny.py

```

# - - -   T a x o n . c o n t a i n s   - - -

def contains ( self, other ):
    """Does self contain other?
    """

    #-- 1 --
    # [ ancestor := nearest ancestor of self and other ]
    ancestor = self.nearestAncestor()

    #-- 2 --
    if self is ancestor:    return 1
    else:                   return 0

```

7.2. Taxon.nearestAncestor()

Here is the algorithm for finding the nearest common ancestor of two taxa `t1` and `t2`:

1. If `t1` and `t2` have the same depth, see if they are identical. If so, the result is `t1`.
2. If `t1` and `t2` are not at the same depth, replace the deeper with its parent and return to Step 1 (p. 22).
3. At this point, we know that `t1` and `t2` are at the same depth but are not the same taxon. Replace each with its parent and return to Step 1 (p. 22).

This procedure is guaranteed to terminate, assuming that the taxa are nodes in a tree. If all else fails, it will terminate when both taxa rise to the root.

txny.py

```
# - - - Taxon.nearestAncestor - - -

def nearestAncestor ( self, other ):
    """Find the nearest ancestor of self and other.
    """

    #-- 1 --
    t1 = self
    t2 = other

    #-- 2 --
    # [ return the nearest ancestor of t1 and t2 ]
    while t1 is not t2:
        #-- 2 body --
        # [ if t1.rank.depth < t2.rank.depth ->
        #     t1 := parent of t1
        # else if t1.rank.depth > t2.rank.depth ->
        #     t2 := parent of t2
        # else ->
        #     t1 := parent of t1
        #     t2 := parent of t2 ]
        if t1.rank.depth < t2.rank.depth:
            t2 = t2.parent
        elif t1.rank.depth > t2.rank.depth:
            t1 = t1.parent
        else:
            t1 = t1.parent
            t2 = t2.parent

    #-- 3 --
    return t1
```

7.3. Taxon.childContaining(): Find child containing a descendant

This method finds the child of `self` that contains a given descendant `desc`.

txny.py

```
# - - - Taxon.childContaining - - -

def childContaining ( self, desc ):
```

```

    """Find self's child whose subtree contains desc.

    [ desc is a Taxon in self.txny ->
      if desc is a descendant of self ->
        return the child of self containing desc
      else -> raise ValueError ]
    """

```

A little error checking first: if `desc` is no deeper than `self`, it cannot be a descendant of `self`.

txny.py

```

#-- 1 --
if desc.rank.depth <= self.rank.depth:
    raise ValueError, ( "Taxon %s is not a descendant of %s." %
                       (desc, self) )

```

At this point we know that `desc` is deeper than `self`, so it is safe to assume that the parent of `desc` is not `None`. If the parent of `desc` is `self`, then `desc` is the answer—it is the child of `self` containing `desc`.

txny.py

```

#-- 2 --
if desc.parent == self:
    return desc

```

Recursion simplifies the rest of this method: either `desc.parent` is a child of `self`, or a node further up the tree.

txny.py

```

#-- 3 --
# [ if desc.parent is a descendant of self ->
#   return the child of self containing parent
#   else -> raise ValueError ]
return self.childContaining ( desc.parent )

```

7.4. Taxon.__cmp__(): Compare taxa in phylogenetic order

This method is called when the usual Python `cmp()` function is applied to two `Taxon` objects. It orders them using the phylogenetic key.

txny.py

```

# - - - T a x o n . _ _ c m p _ _ - - -

def __cmp__ ( self, other ):
    """Compare two taxa, order them phylogenetically.

    [ self and other are Taxon objects ->
      if self occurs before other in phylogenetic order ->
        return -1
      else if self is the same taxon as other ->
        return 0
      else -> return 1 ]
    """
    return cmp ( self.txKey, other.txKey )

```

7.5. Taxon.__getitem__()

This method is called whenever a Taxon item T is indexed as $T[i]$. It returns the i th child taxon of self if there is one; otherwise it raises `KeyError`.

txny.py

```
# - - - T a x o n . _ _ g e t i t e m _ _ - - -  
  
def __getitem__ ( self, n ):  
    """Get the nth child taxon of self.  
    """  
    return self.__childList[n]
```

7.6. Taxon.__iter__()

Returns an iterator that visits the child taxa of self.

txny.py

```
# - - - T a x o n . _ _ i t e r _ _ - - -  
  
def __iter__ ( self ):  
    """Iterates over the child taxa of self.  
    """  
    for child in self.__childList:  
        yield child  
    raise StopIteration
```

7.7. Taxon.__len__()

Same as `.nChildren()`.

txny.py

```
# - - - T a x o n . _ _ l e n _ _ - - -  
  
def __len__ ( self ):  
    """Return the number of child taxa of self."""  
    return len(self.__childList)
```

7.8. Taxon.__str__()

This method is called to convert a Taxon object to a string. It can be useful for debugging.

txny.py

```
# - - - T a x o n . _ _ s t r _ _ - - -  
  
def __str__ ( self ):  
    """Display self as a string.  
    """  
    return ( "%s [%s: %s] txKey=%s status=%s" %  
            ( self.eng, self.rank.name, self.sci, self.txKey,  
              self.status ) )
```

7.9. Taxon.__init__(): The constructor

Users don't call this constructor directly. Its purpose is to convert XML nodes into Taxon objects. Consequently, its calling sequence looks like this:

```
txny.py
# - - -   T a x o n . _ _ i n i t _ _   - - -
def __init__ ( self, txny, parentTaxon, taxonNode ):
    """Constructor for a Taxon object.
    """
```

Note that this constructor is recursive. It converts not only the given `taxonNode` but all its descendants. Pass it the root `TAXON_N` node, and it will build the entire tree.

The first step is to set up the parts of the object that point elsewhere:

```
txny.py
#-- 1 --
self.txny      = txny
self.parent    = parentTaxon
self.__childList = []
```

Next, we set up the parts of the object that are derived from the XML node we're converting:

```
txny.py
#-- 2 --
# [ self.abbr      := STD_ABBR_A attribute from taxonNode, or None
#                  if there is no such attribute
#   self.eng       := ENG_A attribute from taxonNode
#   self.rank      := rank whose code is the RANK_A
#                  attribute from taxonNode
#   self.sci       := SCI_A attribute from taxonNode
#   self.status    := STATUS_A attribute from taxonNode, or None
#                  if there is no such attribute
#   self.tex       := TEX_A attribute from taxonNode, or None if
#                  there is no such attribute
#   self.txKey     := TX_KEY_A attribute from taxonNode ]
self.abbr      = taxonNode.attrib.get(rnc.STD_ABBR_A, None)
self.eng       = getChildContent ( taxonNode, rnc.ENG_N )
wordList      = self.eng.split()
rankCode      = taxonNode.attrib[rnc.RANK_A]
self.rank      = txny.hier.lookupRankCode ( rankCode )
self.sci       = taxonNode.attrib[rnc.SCI_A]
self.status    = taxonNode.attrib.get(rnc.STATUS_A, None)
self.tex       = getChildContent ( taxonNode, rnc.TEX_NAME_N )
self.txKey     = taxonNode.attrib[rnc.TX_KEY_A]
if ( (len(wordList) == 1) or
      (self.rank.depth < txny.hier.speciesRank().depth) ):
    self.engComma = self.eng
else:
    self.engComma = ( "%s, %s" %
                     (wordList[-1], " ".join(wordList[:-1])) )
```

Finally, we recursively translate all this node's descendants.

```

#-- 3 --
# [ self.__childList += Taxon objects made from taxonNode's
#     children, recursively containing all descendants of
#     those children ]
for childElement in taxonNode.xpath ( rnc.TAXON_N ):
    #-- 3 body --
    # [ self.childList += a Taxon object made from
    #     childElement, recursively containing all descendants
    #     of childElement ]
    self.__childList.append ( Taxon ( self.txny, self, childElement
) )

```

8. The abbr.py module

This module contains assorted functions for dealing with bird codes.

8.1. Prologue

The module starts with a short, pro-forma comment, a few imports, and the exported manifest constants.

```

"""abbr.py: Six-letter bird code system for use with txny.py.

For full documentation, see:
    http://www.nmt.edu/~shipman/xnomo/
"""

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

import re                # Standard Python regular expressions

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

ABBR_L      = 6          # Maximum bird code length
BLANK_ABBR  = " " * ABBR_L  # Blank bird code
RE_ABBR     = re.compile ( # Regex for a bird code
    r'[a-zA-Z]{2,6}' )     # Two to six letters
REL_HYBRID  = "^"        # Relationship codes for hybrid...
REL_PAIR    = "|"        # ...and species pair
REL_SIMPLE  = " "        # Simple (not compound) form
REL_MAP     = {          # Maps rel codes to conventional form
    REL_HYBRID: "x", REL_PAIR: "/", REL_SIMPLE: " " }

#--
# Because REL_HYBRID has special meaning inside a regular expression
# of the form [...], it must not be the first character of the set.

```

```

#--
RE_REL      = re.compile ( # Regex for a relationship code
    r'[%s%s]' % (REL_PAIR, REL_HYBRID) )

#--
# Substitutions for confusing color names
#--

COLOR_SUB_2 = {
    "BLACK":    "BK",      "GRAY":    "GY",
    "BLUE":     "BU",      "GREY":    "GY",
    "BROWN":    "BN",      "GREEN":   "GN" }

COLOR_SUB_3 = {
    "BLACK":    "BLK",    "GRAY":    "GRY",
    "BLUE":     "BLU",    "GREY":    "GRY",
    "BROWN":    "BRN",    "GREEN":   "GRN" }

```

8.2. abbreviate()

This function takes an English name and derives a bird code from it using the rules of the six-letter system.

abbr.py

```

#=====
# Specification functions
#-----
# one-word-rule(word) == word, truncated or blank-padded to
#                       size ABBR_L
#-----
# two-word-rule(w1,w2) ==
#   if w1 matches one of the confusing color names ->
#     (the substitute 3-letter color code) + w2[:3]
#   else ->
#     (w1[:3], padded to 3 with hyphens) +
#     (w2[:3], padded to 3 with spaces)
#-----
# three-word-rule(w1,w2,w3) ==
#   if w1 matches one of the confusing color names ->
#     (the substitute 2-letter color code) +
#     w2[0] + w3[:3]
#   else ->
#     w1[:2]+w2[0]+(w3[:3], padded to 3 with spaces)
#-----
# four-word-rule(w1,w2,w3,wLast) ==
#   w1[0]+w2[0]+w3[0]+(wLast[:3], padded to 3 with spaces)
#-----

# - - -   a b b r e v i a t e   - - -

def abbreviate ( eng ):
    "Form the canonical abbreviation from an English name."

```

```

#-- 1 --
# [ words := (eng), uppercased and trimmed of whitespace
#           at both ends, with all hyphens changed to spaces ]
words = eng.upper().strip().replace ( "-", " " )

#-- 2 --
# [ wordList := words, broken into a list on spaces ]
wordList = words.split()

#-- 3 --
# if wordList has 1 element ->
#   result := one-word-rule ( wordList[0] )
# else if wordList has 2 elements ->
#   result := two-word-rule ( wordList[0], wordList[1] )
# else if wordList has 3 elements ->
#   result := three-word-rule ( wordList[0], wordList[1],
#                               wordList[2] )
# else ->
#   result := four-word-rule ( wordList[0], wordList[1],
#                               wordList[2], wordList[-1] ) ]
if len(wordList) > 3:
    result = rule4 ( wordList[0], wordList[1], wordList[2],
                    wordList[-1] )
elif len(wordList) == 3:
    result = rule3 ( wordList[0], wordList[1], wordList[2] )
elif len(wordList) == 2:
    result = rule2 ( wordList[0], wordList[1] )
else:
    result = wordList[0][:ABBR_L].ljust(ABBR_L)

#-- 4 --
return result

# - - -   r u l e 2   - - -

def rule2 ( w1, w2 ):
    """Implements the two-word rule.

    [ w1 and w2 are strings ->
      return two-word-rule(w1,w2) ]
    """

#-- 1 --
# [ head := first three letters of w1, right-padded to
#           length 3 with hyphens ]
head = w1[:3].ljust(3)

#-- 2 --
# [ if w1 matches a confusing color name ->
#   head := the corresponding 3-letter substitute
#   else -> I ]

```

```

if COLOR_SUB_3.has_key ( w1 ):
    head = COLOR_SUB_3[w1]

#-- 3 --
# [ return (head + (first three letters of w2)), right-padded
#   to ABBR_L with spaces ]
return (head+w2[:3]).ljust(ABBR_L)

# - - -   r u l e 3   - - -

def rule3 ( w1, w2, w3 ):
    """Implements the three-word rule.

    [ w1, w2 and w3 are strings ->
      return three-word-rule(w1,w2,w3) ]
    """

#-- 1 --
# [ head := first 2 characters of w1
#   tail := (first character of w2) + (first three characters of w3)
  ]
    head = w1[:2]
    tail = w2[0] + w3[:3]

#-- 2 --
# [ if w1 matches a confusing color name ->
#   head := the corresponding 2-letter substitute
#   else -> I ]
if COLOR_SUB_2.has_key ( w1 ):
    head = COLOR_SUB_2[w1]

#-- 3 --
return head + tail

# - - -   r u l e 4   - - -

def rule4 ( w1, w2, w3, wLast ):
    """Implements the four-word rule.

    [ if all arguments are strings ->
      return four-word-rule(w1,w2,w3,wLast) ]
    """
    return ( "%s%s%s%s" % ( w1[0], w2[0], w3[0], wLast[:3] ) ).ljust(ABBR_L)

```

8.3. engComma (): Invert an English name

Converts "Great Auk" to "Auk, Great", for example.

abbr.py

```
# - - -   e n g C o m m a
```

```

def engComma ( eng ):
    '''Reverse a name as "GENERIC-PART[, SPECIFIC-PART]"

    [ eng is a string ->
      if eng contains any spaces ->
        return (longest suffix of eng not containing a space) +
          ", " + (eng up to last space)
      else -> return eng ]
    ...
    #-- 1 --
    # [ if eng contains any spaces ->
    #   spacePos := position of the last space
    #   else ->
    #     return eng ]
    spacePos = eng.rfind ( ' ' )
    if spacePos < 0:
        return eng

    #-- 2 --
    return "%s, %s" % (eng[spacePos+1:], eng[:spacePos])

```

8.4. engDeComma (): Normalize an English name

Converts "Heron, Great Blue" to "Great Blue Heron", for example.

abbr.py

```

# - - -   e n g D e C o m m a

def engDeComma ( eng ):
    '''Transform "GENERIC-PART, SPECIFIC-PART" to the normal order.
    ...
    #-- 1 --
    # [ if eng contains a comma ->
    #   commaPos := its position
    #   else ->
    #     return eng ]
    commaPos = eng.find ( ',' )
    if commaPos < 0:
        return eng

    #-- 2 --
    return ( "%s %s" %
             (eng[commaPos+1:].strip(), eng[:commaPos]) )

```

8.5. class BirdId

Each instance of this class represents one simple or compound bird identity.

abbr.py

```

# - - - - -   c l a s s   B i r d I d   - - - - -

class BirdId:

```

```
"""Represents a single bird code, or two codes with a relationship.
```

```
Exports:
```

```
BirdId ( txny, abbr, rel=None, abbr2=None, q=None ):  
    [ (txny is a Txny object) and  
      (abbr is the first or only bird code) and  
      (rel is REL_SIMPLE, REL_HYBRID, or REL_PAIR) and  
      (abbr2 is None for rel=REL_SIMPLE, or the second  
      bird code for rel != REL_SIMPLE) and  
      (q is None normally or "?" for questionable ID) ->  
      return a new BirdId representing that bird  
      identification ]  
.txny:          [ as passed to constructor, read-only ]  
.abbr:          [ as passed to constructor, read-only ]  
.rel:          [ as passed to constructor, read-only ]  
.abbr2:        [ as passed to constructor, read-only ]  
.q:            [ as passed to constructor, read-only ]  
.fullAbbr:  
    [ if self.rel == REL_SIMPLE -> self.abbr.rstrip()  
      else ->  
        self.abbr.rstrip() + self.rel + self.abbr2.rstrip() ]  
.taxon:  
    [ a Taxon representing the smallest taxon in self.txny  
      that contains abbr and, if present, abbr2 ]  
.normalize():  
    [ self.abbr, uppercased and blank-padded to size ABBR_L ]  
.__str__():  
    [ return self's English name ]  
BirdId.scan ( txny, scan ):      # Static method  
    [ (txny is a Txny object) and (scan is a Scan object) ->  
      if the current line of scan starts with a simple or  
      compound bird code ->  
        scan := scan advanced past that code  
        return a new BirdId object representing that code  
      else ->  
        scan := scan advanced past valid-looking parts, if any  
        scan += error message  
        raise ValueError ]  
BirdId.scanFlat ( txny, scan ):  # Static method  
    [ (txny is a Txny object) and (scan is a Scan object) ->  
      if the current line of scan starts with a  
      compound bird code in flat-field format ->  
        scan := scan advanced past that code  
        return a new BirdId object representing that code  
      else ->  
        scan := scan advanced past valid-looking parts, if any  
        scan += error message  
        raise ValueError ]  
BirdId.scanAbbr ( txny, scan ):  # Static method  
    [ (txny is a Txny object) and (scan is a Scan object) ->  
      if scan starts with a bird code valid in txny ->  
        scan := scan advanced past that code  
        return that code
```

```

        else ->
            scan := scan advanced valid-looking parts, if any
            scan += error message
            raise ValueError ]
    BirdId.scanAbbrFlat ( txny, scan ): # Static method
    [ (txny is a Txny object) and (scan is a Scan object) ->
        if scan starts with a bird code valid in txny,
        right blank-padded to length ABBR_L ->
            scan := scan advanced past that code
            return the code without its blank padding
        else ->
            scan := scan advanced past valid-looking stuff if any
            scan += error message
            raise ValueError ]

    State/Invariants:
    self.abbr < self.abbr2
    """

```

8.6. BirdId.__init__()

This the constructor for a BirdId object.

abbr.py

```

#=====
# Specification functions
#-----
# normalize(code) ==
# code, uppercased and right-padded to length ABBR_L
#-----

# - - - B i r d I d . _ _ i n i t _ _ - - -

def __init__ ( self, txny, abbr, rel=None, abbr2=None, q=None ):
    """Constructor for BirdId objects"""

    #-- 1 --
    self.txny = txny

```

Next, we store the code or codes in normalized form. If there are two codes, we store the lesser in .abbr and the greater in .abbr2.

abbr.py

```

#-- 2 --
# [ if rel is None or REL_SIMPLE ->
#   self.abbr := normalize(abbr)
#   self.rel := None
# else ->
#   self.rel := rel
#   self.abbr := min(normalize(abbr), normalize(abbr2))
#   self.abbr2 := max(normalize(abbr), normalize(abbr2)) ]
if ( ( rel is None ) or ( rel == REL_SIMPLE ) ):
    self.abbr = self.normalize ( abbr )

```

```

        self.rel    = None
        self.abbr2 = None
    else:
        self.rel = rel
        abbrList = [ self.normalize(abbr),
                    self.normalize(abbr2) ]
        abbrList.sort()
        self.abbr, self.abbr2 = abbrList

```

If the `q` argument is false (`None` or the empty string), we set `self.q` to the empty string; otherwise we save its value in `self.q`.

abbr.py

```

#-- 3 --
# [ if q is false ->
#     self.q = ""
#     else ->
#         self.q = q ]
self.q = q or ""

```

Next we build up the `.fullAbbr` attribute:

abbr.py

```

#-- 4 --
if self.rel is None:
    self.fullAbbr = "%s%s" % (self.abbr, self.q)
else:
    self.fullAbbr = ( "%s%s%s%s" %
                     (self.abbr.rstrip(), self.rel,
                     self.abbr2.rstrip(), self.q) )

```

Lastly, we set up the `.taxon` attribute by finding the nearest common ancestor of the two forms. In the process, we will also need to look up the taxa for one or both codes. If there is only one code, use its `taxon`.

abbr.py

```

#-- 5 --
# [ if self.rel is None ->
#     self.taxon := self.txny's Taxon for self.abbr
#     else ->
#         self.taxon := nearest common ancestor of self.abbr
#                     and self.abbr2 in self.txny ]
if self.rel is None:
    self.taxon = self.txny.lookupAbbr ( self.abbr )
else:
    taxon1      = self.txny.lookupAbbr ( self.abbr )
    taxon2      = self.txny.lookupAbbr ( self.abbr2 )
    self.taxon  = taxon1.nearestAncestor ( taxon2 )

```

8.7. BirdId.normalize()

This function normalizes a bird code, uppercasing it and right-padding with blanks if necessary.

abbr.py

```

# - - -   B i r d I d . n o r m a l i z e   - - -

```

```
def normalize ( self, abbr ):
    """Normalize a bird code."""
    return abbr.upper().ljust(ABBR_L)
```

8.8. `birdId.engComma()`: Inverted English name

This method returns the English name associated with the form, except that each name has the form "last, first".

abbr.py

```
# - - -   B i r d I d . e n g C o m m a   - - -

def engComma ( self ):
    """Return self's name in 'last, first' order.
    """
    if self.rel is None:
        eng = self.txny.abbrToEng ( self.abbr )
        commafied = BirdId.commafy ( eng )
        return "%s%s" % (commafied, self.q)
    else:
        eng1 = self.txny.abbrToEng ( self.abbr )
        com1 = BirdId.commafy ( eng1 )
        eng2 = self.txny.abbrToEng ( self.abbr2 )
        com2 = BirdId.commafy ( eng2 )
        return ( "%s %s %s%s" %
                (com1, REL_MAP[self.rel], com2, self.q) )
```

8.9. `BirdId.commafy()`: Invert word order (static method)

This handy method rearranges an English name so that the last word comes first, followed by a comma, followed by the rest of the name.

abbr.py

```
# - - -   B i r d I d . c o m m a f y   - - -

# @staticmethod
def commafy ( eng ):
    """Transform 'Great Blue Heron' -> 'Heron, Great Blue'

    [ eng is a string ->
      if eng has no internal spaces or ends with " sp." ->
        return eng
      else ->
        return (last word of eng)+", "+(other words of eng) ]
    """
    #-- 1 --
    if eng.endswith ( ' sp.' ):
        return eng

    #-- 2 --
    # [ wordList := eng, broken at each region of whitespace ]
    wordList = eng.split()
```

```

#-- 3 --
# [ if len(wordList) < 2 ->
#     return eng
#     else ->
#         return wordList[-1] + ", " + (wordList[:-1], joined
#             by spaces ]
if len(wordList) < 2:
    return eng
else:
    result = ( "%s, %s" %
                (wordList[-1], " ".join(wordList[:-1])) )
    return result

commafy = staticmethod(commafy)

```

8.10. BirdId.__str__()

Displays a BirdId object in string form.

abbr.py

```

# - - -   B i r d I d . _ _ s t r _ _   - - -

def __str__ ( self ):
    """Display the English name of self.
    """

    #-- 1 --
    # [ if self.rel is None ->
    #     return the .eng attribute of self.abbr's Taxon
    #     else ->
    #         return the .eng attributes of the Taxon objects of
    #         self.abbr and self.abbr2, separated by self.rel in
    #         conventional form ]
    if self.rel is None:
        eng = self.txny.abbrToEng ( self.abbr )
        return "%s%s" % (eng, self.q)
    else:
        eng1 = self.txny.abbrToEng ( self.abbr )
        eng2 = self.txny.abbrToEng ( self.abbr2 )
        return ( "%s %s %s%s" %
                ( eng1, REL_MAP[self.rel], eng2, self.q ) )

```

8.11. BirdId.scan()

Using the author's Scan object for parsing an input stream, this static method looks for a simple or compound bird code.

abbr.py

```

# - - -   B i r d I d . s c a n   - - -

def scan ( txny, scan ):
    """Scan a simple or compound bird code.
    """

```

First we check to see if the beginning of the line matches the regular expression RE_ABBR. If not, we can fail right away.

abbr.py

```
#-- 1 --
# [ if scan starts with a simple bird code ->
#   scan := scan advanced past matching characters
#   abbr := matching characters
#   rel := None
#   abbr2 := None
# else ->
#   scan += error message
#   raise ValueError ]
abbr = BirdId.scanAbbr ( txny, scan )
rel = abbr2 = None
```

If the next character is a relationship code, it must be followed by a second simple bird code.

abbr.py

```
#-- 2 --
# [ if scan starts with a relationship code followed by
#   a valid simple bird code ->
#   scan := scan advanced past all that
#   rel := the relationship code
#   abbr2 := the simple bird code
# else if scan starts with a relationship code not
#   followed by a valid simple bird code ->
#   scan += error message
#   raise ValueError
# else -> I ]
m = scan.tabReMatch ( RE_REL )
if m is not None:
#-- 2.1 --
# [ if scan starts with a valid simple bird code ->
#   rel := matched string from m
#   scan := scan advanced past the bird code
# else ->
#   scan += error message
#   raise ValueError ]
rel = m.group()
abbr2 = BirdId.scanAbbr ( txny, scan )
```

The next character may be "?"; in that case set the q attribute to that string.

abbr.py

```
#-- 3 --
# [ if scan starts with "?" ->
#   scan := scan advanced 1
#   q := "?"
# else ->
#   q := "" ]
if scan.tabMatch ( "?" ):
    q = "?"
else:
    q = ""
```

Finally we bundle all the parts together as a `BirdId` instance.

abbr.py

```
#-- 4 --
return BirdId ( txny, abbr, rel, abbr2, q )
```

The next line is necessary to make this method static.

abbr.py

```
scan = staticmethod(scan)
```

8.12. `BirdId.scanFlat()`

This is similar to the `.scan()` method, but assumes a fixed-field format.

abbr.py

```
# - - -   B i r d I d . s c a n F l a t   - - -

def scanFlat ( txny, scan ):
    """Scan a flat-field compound bird code.
    """
```

First we require a bird code, followed by enough spaces to make it length `ABBR_L`.

abbr.py

```
#-- 1 --
# [ if scan starts with a bird code as a flat field ->
#   scan := scan advanced past that code
#   abbr := that code
# else ->
#   scan += error message
#   raise ValueError ]
abbr = BirdId.scanAbbrFlat ( txny, scan )
```

Next comes the relationship code.

abbr.py

```
#-- 2 --
# [ if scan starts with REL_SIMPLE followed by BLANK_ABBR ->
#   scan := scan advanced past all that
#   return a new BirdId made from txny and abbr
# else -> I ]
m = scan.tabMatch ( REL_SIMPLE )
if m is not None:
    abbr2 = scan.tabMatch ( BLANK_ABBR )
    if abbr2 is not None:
        return BirdId ( txny, abbr )
    else:
        message = ( "The second bird code must be blank when "
                    "the rel-code is blank." )
        scan.error ( message )
        raise ValueError, message
```

Since the rel-code isn't blank, we must now parse a valid rel-code followed by a valid bird code.

abbr.py

```
#-- 3 --
# [ if scan starts with a rel-code followed by a bird code ->
```

```

# scan := scan advanced past all that
# rel := the rel-code
# abbr2 := the bird code
# else ->
# scan += error message
# raise ValueError ]
m = scan.tabReMatch ( RE_REL )
if m is None:
    message = "Expecting a relationship code."
    scan.error ( message )
    raise ValueError, message
rel = m.group()
abbr2 = BirdId.scanAbbrFlat ( txny, scan )

#-- 3 --
return BirdId ( txny, abbr, rel, abbr2 )

scanFlat = staticmethod(scanFlat)

```

8.13. BirdId.scanAbbr()

This is a helper method used by `.scan()` and `.scanFlat()`. It is a static method.

abbr.py

```

# - - - B i r d I d . s c a n A b b r - - -

def scanAbbr ( txny, scan ):
    """Scan one bird code.
    """

```

We use the regular expression `RE_ABBR` to match any leading characters in the `scan` object. If it matches, it returns a `Match` object whose `.group()` method returns the matching text. If it returns `NONE`, there is no match.

abbr.py

```

#-- 1 --
# [ if scan starts with characters matching RE_ABBR ->
# scan := scan advanced past matching characters
# m := a Match object representing the matched text
# else ->
# m := None ]
m = scan.tabReMatch ( RE_ABBR )

#-- 2 --
# [ if m is None ->
# scan += error message
# raise ValueError
# else ->
# abbr := m.group()
if m is None:
    scan.error ( "Expecting a bird code." )
    raise ValueError, "Expecting a bird code."
else:
    abbr = m.group()

```

```

#-- 3 --
# [ if abbr is defined in txny ->
#     return abbr
#     else -> raise ValueError ]
try:
    taxon = txny.lookupAbbr ( abbr )
    return abbr
except KeyError, detail:
    message = "Unknown bird code '%s'" % abbr
    scan.error ( message )
    raise ValueError, message

```

Finally, make this a static method.

abbr.py

```
scanAbbr = staticmethod ( scanAbbr )
```

8.14. BirdId.scanAbbrFlat()

This scans a “flat file” bird code, consisting of a bird code right-padded with spaces to length `ABBR_L`.

abbr.py

```

# - - -   B i r d I d . s c a n A b b r F l a t   - - -

def scanAbbrFlat ( txny, scan ):
    """Scan a bird code from a flat file record.
    """

    #-- 1 --
    # [ if scan starts with a bird code valid in txny ->
    #     scan := scan advanced past that code
    #     abbr := that code
    #     else ->
    #     scan := scan advanced valid-looking parts, if any
    #     scan += error message
    #     raise ValueError ]
    abbr = BirdId.scanAbbr ( txny, scan )

    #-- 2 --
    # [ if (len(abbr) == ABBR_L) ->
    #     return abbr
    #     else ->
    #     pad := a string of (ABBR_L - len(abbr)) spaces ]
    if len(abbr) == ABBR_L:
        return abbr
    else:
        pad = " " * (ABBR_L - len(abbr))

    #-- 3 --
    # [ if scan starts with pad ->
    #     return abbr
    #     else ->
    #     scan += error message

```

```

#     raise ValueError ]
matcher = scan.tabMatch ( pad )
if matcher is None:
    message = ( "Expecting a pad of %d spaces after "
                "a bird code." % len(pad) )
    scan.error ( message )
    raise ValueError, message
else:
    return abbr

scanAbbrFlat = staticmethod ( scanAbbrFlat)

```

8.15. BirdId.parse(): Parse a code from a string

This static method takes a simple or compound code as a string and returns a `BirdId`.

abbr.py

```

# - - -   B i r d I d . p a r s e   - - -

def parse ( txny, s ):
    """Parse a bird code from a string.
    """

```

First we check to see if the last character is "?", and remove it and set `q` to "?" if so. Otherwise we set `q` to `None`.

abbr.py

```

#-- 1 --
if s[-1] == '?':
    s = s[:-1]
    q = '?'
else:
    q = None

```

The general approach here is to use regular expressions to recognize the first or only bird code, optionally followed by a relationship code and a second bird code.

abbr.py

```

#-- 2 --
# [ if s starts with pattern RE_ABBR ->
#     abbr := matching part of s
#     abbr2 := None
#     rel := None
# else -> raise ValueError ]
m = RE_ABBR.match ( s )
if m is None:
    raise ValueError, ( "Not a valid BirdId: '%s'" % s )
abbr = m.group()
rest = s [ m.end() : ]

#-- 3 --
# [ if rest is empty ->
#     return a new simple BirdId made from abbr
# else -> I ]

```

```

if len(rest) == 0:
    return BirdId ( txny, abbr, None, None, q )

#-- 4 --
# [ if rest starts with pattern RE_REL ->
#     rel     := the matching part of rest
#     final   := rest past the matching part
#     else -> raise ValueError ]
m = RE_REL.match ( rest )
if m is None:
    raise ValueError, ( "Expecting a relationship code:"
        "'%s'" % rest )
rel = m.group()
final = rest [ m.end() : ]

#-- 5 --
# [ if final starts with pattern RE_ABBR ->
#     abbr2   := matching part
#     tail    := part of final past the match
#     else -> raise ValueError ]
m = RE_ABBR.match ( final )
if m is None:
    raise ValueError, ( "Expecting 2nd abbr: '%s'" %
        final )
abbr2 = m.group()
tail = final [ m.end() : ]

#-- 6 --
if len(tail.strip()) > 0:
    raise ValueError, ( "Garbage after 2nd abbr: '%s'" %
        tail )
else:
    return BirdId ( txny, abbr, rel, abbr2, q )

parse = staticmethod ( parse )

```

