

archx: A program to index a photo archive



John W. Shipman

2009-10-10 18:35

Table of Contents

1. Overview	1
1.1. How to get this document	2
2. Operation of the <code>archx.py</code> script	2
3. The <code>archx.rnc</code> schema	3
4. <code>archx.py</code> : The script	4
4.1. Prologue	4
4.2. Imports	4
4.3. Manifest constants	5
4.4. Main	6
4.5. <code>processArchive()</code> : Process the contents of one archive	7
4.6. <code>processFile()</code> : Check one image file	9
4.7. <code>writeIndex()</code> : Output the index for one archive directory	11
4.8. <code>class Imagex</code> : An object to describe one image	12
4.9. <code>Imagex.__init__()</code> : Constructor	13
4.10. <code>Imagex.writeNode()</code> : Translate self to XML	14
4.11. Epilogue	14
5. <code>archindex.py</code> : Reader for archive index files	14
5.1. Prologue	14
5.2. <code>class ArchiveIndex</code> : Index of one archive	15
5.3. <code>ArchiveIndex.__init__()</code> : Constructor	16
5.4. <code>ArchiveIndex.getArchImage()</code> : Retrieve an archived image	16
5.5. <code>ArchiveIndex.genArchImages()</code> : Generate contained image entries	17
5.6. <code>ArchiveIndex.addArchImage()</code> : Add one image	17
5.7. <code>ArchiveIndex.readFile()</code> : Instantiate from XML	17
5.8. <code>class ArchImage</code> : Archived catalog entry	19
5.9. <code>ArchImage.readNode()</code> : Convert an XML node	19
5.10. <code>getIntAttr()</code> : Retrieve an integer attribute value	20
6. Defect statistics	21
6.1. Syntax errors	21
6.2. Strong typing errors	21
6.3. Logic errors	22

1. Overview

The program described in this document is part of a system for indexing photos. For documentation on the base catalog, see *An XML-based bird image cataloging system*¹.

¹ <http://infohost.nmt.edu/~john/scans/slides/ims/>

As scans are accumulated, they must be archived on CDs. The current scheme is to place groups of reference PNG files into directories named `bird-001`, `bird-002`, and so on, each sized to fit on a CD.

The purpose of the **archx** program is to build an index of one of these CD-sized directories. Given an archive number (e.g., `013` for archive directory `bird-013`), it finds all the `.png` image files in that directory, and does these operations for each image:

1. The image file's base name should be its catalog number. If that number is not defined in the `bird-images.xml` file, an error message is written.
2. The program reads the image file and determines its image height and width in pixels. We can calculate the actual image size using the `SCAN` attribute in the image's catalog entry.
3. If there is no thumbnail image in the standard thumbnail directory, one is created.

A file named `arch-nnn.xml` will be written by this script, describing all the images in that file. This file will be used as input to a later processing stage that builds a set of web pages displaying the images of each species (yet to be written).

1.1. How to get this document

This publication is available in Web form² and also as a PDF document³. Please forward any comments to `john@nmt.edu`.

Files mentioned in this document include:

- `archx.rnc`: The schema for the output file in Relax RNG Compact Syntax.
- `archx.py`: The actual script.
- `dom_helpers.py`: The author's helper routines for use with the DOM (XML Document Object Model).
- For `birdimages.py`, see *An XML-based bird image cataloging system*⁴.

2. Operation of the `archx.py` script

The `archx.py` script takes one or more arguments: each is an archive number.

Let's call the archive number `nnn`. The script assumes the current directory has this structure:

- The image archive directory is subdirectory "`bird-nnn/`".
- The thumbnails live in subdirectory "`thumb/`".
- The `arch-nnn.xml` file will be written in subdirectory "`indices/`".

For example, this command

```
archx.py 006
```

would index all the `.png` files in `bird-006/`, produce thumbnails for them in `thumb/`, and write its index to `indices/arch-006.xml`.

² <http://www.nmt.edu/~john/scans/slides/archx/>

³ <http://www.nmt.edu/~john/scans/slides/archx/archx.pdf>

⁴ <http://www.nmt.edu/~john/scans/slides/slidecat/>

3. The archx.rnc schema

This section describes the XML schema for the index files created by **archx**. This schema uses the Relax NG Compact Syntax; see Relax NG Compact Syntax (RNC)⁵ for a description of the schema language.

The schema is presented here in “literate programming” style. For more information on literate programming, see *A source extractor for lightweight literate programming*⁶.

First is a comment block that points back here to the documentation:

```
# archx.rnc: Relax NG schema for bird slide archive index.
# For documentation, see:
#   http://www.nmt.edu/~john/slides/archx/
#-----
```

archx.rnc

The goal symbol is `archive-index`, which represents the index of one entire archive directory (one CD's worth).

```
start = archive-index
archive-index = element archive-index
{ image*
}
```

archx.rnc

Each image is described by one `image` element:

```
image = element image
{ attribute cat-no { text },
  attribute high { xsd:positiveInteger },
  attribute wide { xsd:positiveInteger }
}
```

archx.rnc

The attributes are:

cat-no

The image's catalog number. This must match a `cat-no` attribute in some `original` element in the `birdimages.xml` file.

high

The image's height in millimeters, to a precision of 0.1mm.

wide

The image's width in pixels.

Here's a small sample file:

```
<archive-index>
  <image cat-no="2005-09-05-0022" wide="512" high="462"/>
  <image cat-no="2005-02-18a0005" wide="2018" high="2796"/>
</archive-index>
```

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

⁶ <http://www.nmt.edu/tcc/help/lang/python/examples/litsource/>

4. archx.py: The script

The actual archx.py script follows.

4.1. Prologue

The script starts with a comment block pointing back here to the documentation, and two variables for the program name and version number.

```
archx.py
#!/usr/local/bin/python
#=====
# archx.py: Index one archive directory of bird images.
#   For documentation, see:
#   http://www.nmt.edu/~john/slides/archx/
#-----
PROGRAM_NAME      = "archx.py"
EXTERNAL_VERSION  = "0.0"
```

4.2. Imports

Next comes imports. First, standard Python modules. We need `sys` to get the command line arguments and standard streams. We also need `os` to read directories.

```
archx.py
#=====
# Imports
#-----

import sys
import os
```

The Python Imaging Library deals with images: it can size an image and make a thumbnail. For more information, see *Python Imaging Library (PIL)*⁷.

```
archx.py
import Image
```

Next we'll need the author's module for generating XML. For sources and documentation, see *Python and the XML Document Object Model (DOM) with 4Suite*⁸.

```
archx.py
import xml4create as xc
```

The `birdimages.py` module is an interface to the `birdimages.xml` file that allows us to look up catalog numbers.

```
archx.py
import birdimages
```

The next import needs a little explanation. In the code that refers to XML, we don't want to use string constants for element or attribute names like `image` and `cat-no`. Preferred practice is to declare a global, manifest constant for each element or attribute name, so that if the schema changes, we can rapidly locate all references to the changed name. So we use a tool named *pyrang* that extracts all the

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

⁸ <http://www.nmt.edu/tcc/help/pubs/pyxml4/>

element and attribute names from a schema, and generates Python assignment statements for each one. See *pyrang: A single-sourcing tool for Python-XML applications*⁹. The generated file is named `rnc_archx.py`, and the generated names are prefixed with the string “**RNC_**”, and suffixed with “**_N**” for element names and “**_A**” for attribute names. For example, the name of the `image` element is `RNC_IMAGE_N`, and the name of the `cat - no` attribute is `RNC_CAT_NO_A`.

archx.py

```
from rnc_archx import *
```

4.3. Manifest constants

This section defines constants used throughout the script.

archx.py

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

4.3.1. ARCHIVE_PREFIX

This string is prefixed to the archive number to get the name of the archive directory.

archx.py

```
ARCHIVE_PREFIX = "bird-"
```

4.3.2. BIRD_CATALOG_NAME: Name of the bird images catalog

This is the name of the XML file representing the catalog of bird images that we check to insure all archived images are cataloged.

archx.py

```
BIRD_CATALOG_NAME = "birdimages.xml"
```

4.3.3. INDEX_DIR: Archive index directory name

archx.py

```
INDEX_DIR = "indices/"
```

4.3.4. THUMB_DIR

Name of the subdirectory where thumbnail images are written.

archx.py

```
THUMB_DIR = "thumb/"
```

4.3.5. THUMB_WIDE: Width of thumbnails

Maximum width of a thumbnail image in pixels.

archx.py

```
THUMB_WIDE = 200
```

⁹ <http://www.nmt.edu/tcc/help/lang/python/examples/pyrang/>

4.3.6. THUMB_HIGH: Height of thumbnails

Maximum height of a thumbnail image in pixels.

archx.py

```
THUMB_HIGH = 200
```

4.3.7. THUMB_EXTENSION: File type for thumbnail images

This is the filename extension that determines what image file type is written in the thumbnail directory.

archx.py

```
THUMB_EXTENSION = ".jpg"
```

4.4. Main

Note

This script is written under a blanket precondition that we have write access to the thumbnail directory, THUMB_DIR, and the index directory, INDEX_DIR.

Here is the main program, including an intended function for the script as a whole.

archx.py

```
#=====
# Functions and classes
#-----

# - - -   m a i n   - - -

def main():
    """Main program.

    [ let
      archive-set == set of archive directories named
        in command line arguments
      file-set == set of image files in archive directories
        named in command line arguments
    in:
      sys.stdout += report listing files in file-set
        that appear to be bird images but are not in
        the bird image catalog
      thumbnail directory := thumbnail directory with
        thumbnail images made from file-set
      index directory := index directory with arch-NNN.xml
        files added describing archive-set ]

    """
```

First we must read the bird image catalog, which will be used to verify that all the archived images are properly cataloged. The `.readFile()` method is a static method in class `ImageCatalog` that reads the XML serialization of the image catalog and returns it as an `ImageCatalog` object.

```

#-- 1 --
# [ BIRD_CATALOG_NAME is a readable file valid against
#   birdimages.rnc ->
#       imageCatalog := an ImageCatalog object representing
#       BIRD_CATALOG_NAME ]
imageCatalog = birdimages.ImageCatalog.readFile (
                BIRD_CATALOG_NAME )

```

All that remains is to step through the archive names given as command line arguments, and process each one. See Section 4.5, “processArchive(): Process the contents of one archive” (p. 7).

```

#-- 2 --
for archNo in sys.argv[1:]:
    #-- 2 body --
    # [ sys.stdout += report listing image files in
    #     archive (archNo) that appear to be bird images
    #     but are not in the bird image catalog
    #     thumbnail directory := thumbnail directory with
    #     thumbnail images made image files in archive (archNo)
    #     index directory := index directory with arch-(archNo).xml
    #     files added describing image files in archive (archNo) ]
    processArchive ( imageCatalog, archNo )

```

4.5. processArchive(): Process the contents of one archive

This function does all the processing for one archive directory full of images.

```

# - - -   p r o c e s s A r c h i v e   - - -
def processArchive ( imageCatalog, archNo ):
    """Process one archive directory.

    [ (imageCatalog is a birdimages.ImageCatalog instance) and
      (archNo is the numeric part of an archive directory name) ->
      sys.stdout += report listing image files in
        archive (archNo) that appear to be bird images
        but are not in imageCatalog
      thumbnail directory := thumbnail directory with
        thumbnail images made image files in archive (archNo)
      index directory := index directory with an
        arch-(archNo).xml file added describing image files
        in archive (archNo) ]

    """

```

The argument is the archive number, which must be appended to ARCHIVE_PREFIX to get the name of the archive directory. We also allocate an empty list `imagexList` that will accumulate descriptions of each bird image as a sequence of `Imagex` objects; see Section 4.8, “class Imagex: An object to describe one image” (p. 12).

```

#-- 1 --
# [ archDir := directory name for archive (archNo )
#   imagexList := a new, empty list ]
archDir = ARCHIVE_PREFIX + archNo
imagexList = []

```

We use the standard library `os.listdir()` function to get a list of all the files in that directory. Just for neatness, we'll then sort it.

```

#-- 2 --
# [ fileList := list of files in directory (archDir), sorted
fileList = os.listdir ( archDir )
fileList.sort()

```

For each file name in `fileList`, we call Section 4.6, “`processFile()`: Check one image file” (p. 9) to perform the processing steps for that file, including the generation of an `Imagex` object that will hold the information we need to write the index file.

```

#-- 3 --
# [ imagexList += Imagex objects representing the
#   files in fileList that are valid bird images and
#   indexed in imageCatalog
#   sys.stdout += report of bird images in fileList that are
#   not in imageCatalog ]
for fileName in fileList:
#-- 3 loop --
# [ if (archDir+fileName) is a bird image in imageCatalog ->
#   imagexList += an Imagex object representing
#   that image
#   thumbnail directory += a thumbnail of that image
# else if (archDir+fileName) is a bird image but not
# in imageCatalog ->
#   sys.stdout += (message about uncataloged image)
# else -> I ]

```

The body of the loop has three steps. First we build the relative path to the image file. Then we call Section 4.6, “`processFile()`: Check one image file” (p. 9), which returns an `Imagex` object if everything went okay; otherwise it returns `None`. Then if the return value is not `None`, we can append it to `imagexList`. See Section 4.6, “`processFile()`: Check one image file” (p. 9).

```

#-- 3.1 --
# [ pathName := archDir + fileName ]
pathName = os.path.join ( archDir, fileName )

#-- 3.2 --
# [ if pathName is a bird image in imageCatalog ->
#   thumbnail directory += a thumbnail of that
#   image
#   result := an Imagex object representing that
#   image
# else if pathName is a bird image not in imageCatalog
# or not a bird image ->

```

```

#      sys.stdout += error message
#      result := None
# else ->
#      result := None ]
result = processFile ( imageCatalog, pathName )

#-- 3.3 --
if result is not None:
    imagexList.append ( result )

```

All that remains is to write the index file for this archive. Needed for Section 4.7, “writeIndex(): Output the index for one archive directory” (p. 11) are two items: the archive number, and the list `imagexList` containing the details of the valid, cataloged images.

archx.py

```

#-- 4 --
# [ imagexList is a list of Imagex objects ->
#   index directory := index directory with an
#   (archDir+".xml") file added representing imagexList ]
writeIndex ( archDir, imagexList )

```

4.6. processFile(): Check one image file

This function is called to check one file name. If the file isn't a bird image, it gets ignored. If its name resembles that of a bird image, it is checked to make sure it's in the catalog.

archx.py

```

# - - - p r o c e s s F i l e - - -
def processFile ( imageCatalog, pathName ):
    """Check one file and, if valid, return an Imagex object.

    [ (imageCatalog is an ImageCatalog) and
      (pathName is a nonempty string) ->
        if (pathName looks like a bird image name) and
          (pathName is a catalog number in ImageCatalog) and
          (pathname names a readable image file) ->
            thumbnail directory += a thumbnail of that image
            return an Imagex object representing that image
        else if (pathName looks like a bird image name) and
          ((pathName is not a catalog number in ImageCatalog) or
           (pathname does not name a readable image file)) ->
            sys.stdout += error message
            return None
        else ->
            return None ]
    """

```

First we disassemble the full path name of the image file, saving its file name (minus the extension) in `baseName`.

archx.py

```

#-- 1 --
# [ baseName := pathName minus its path component and

```

```
#             file extension ]
dirPath, fileName = os.path.split ( pathName )
baseName, extension = os.path.splitext ( fileName )
```

At this writing, all images have one of two file name formats. Bird images have a year-month-day format:

```
yyyymmddxnn
```

The *nn* part is the film frame number. The *x* character is usually a period, but can be a lowercase letter when there are multiple images on the same day with the same frame number.

Nonbird images are prefixed with the letter “n”:

```
nyyyymmddxnn
```

So at this point in time, the test for whether a file represents a bird image is to see whether it starts with a digit. If not, we can just return `None`; no error checking is done on nonbird images.

There is one extra wrinkle. If the filename is a “hidden file” starting with ‘.’, `baseName` will be the empty string. In that case the file is clearly not an image file.

archx.py

```
## 2 --
# [ if baseName is nonempty and starts with a letter ->
#     I
# else -> return None ]
if ( ( len(baseName) == 0 ) or
      ( not ( baseName[0].isdigit() ) ) ):
    return None
```

Next we check to see if the image has been cataloged. If not, we write an error message and return `None`.

archx.py

```
## 3 --
# [ if baseName is a catalog number in imageCatalog ->
#     I
# else ->
#     sys.stdout += error message
#     return None ]
try:
    orig = imageCatalog.getOriginal ( baseName )
except KeyError:
    print "*** Uncataloged: %s" % pathName
    return None
```

Next we use the `Image` class constructor to read the image file and extract the width and height. See Section 4.8, “class `Image`: An object to describe one image” (p. 12).

archx.py

```
## 4 --
# [ if pathName names a readable, valid image file ->
#     result := an Image object representing that image
# else -> raise IOError ]
result = Image ( pathName )
```

One more task remains: creation of the thumbnail image. The path name of the thumbnail is `THUMB_DIR+pathName`. Converting the full-sized image to a thumbnail is a single method call on the

Image object: the `.thumbnail()` method takes a 2-tuple specifying the maximum width and height, and the aspect ratio is preserved.

Technically, we have broken the encapsulation of the `ImageX` object by replacing its `.image` attribute with a different image. However, since that attribute is not used for anything except writing the index file after this, no harm is done. Also, the steps we would need to take to avoid this are computationally expensive: we would need to make a copy of the entire image (many of which run into the tens of megabytes) before reducing it to a thumbnail.

archx.py

```
#-- 5 --
# [ thumbPath := THUMB_DIR + baseName + THUMB_EXTENSION
#   result := result with its image replaced by a
#         thumbnail no larger than (THUMB_WIDE, THUMB_HIGH) ]
thumbPath = "%s%s%s" % (THUMB_DIR, baseName, THUMB_EXTENSION)
result.image.thumbnail ( (THUMB_WIDE, THUMB_HIGH) )
```

Finally, assuming that we can, we write the thumbnail image. This could fail, but it falls under the blanket precondition that we have write access to the thumbnail directory. Assuming all that works, we can then return the `ImageX` result to the caller.

archx.py

```
#-- 6 --
# [ if thumbPath names a file that can be created new ->
#   that file := result.image with its type determined
#             by thumbPath's extension
#   else -> raise IOError ]
result.image.save ( thumbPath )

#-- 7 --
return result
```

4.7. writeIndex(): Output the index for one archive directory

This function writes an XML file conforming to the `archx.rnc` schema (see Section 3, “The `archx.rnc` schema” (p. 3)).

archx.py

```
# - - -   w r i t e I n d e x   - - -

def writeIndex ( archDir, imagexList ):
    """Generate the XML index file.

    [ (archNo is an archive directory name as a string) and
      (imagexList is a list of ImageX objects) ->
      index directory := index directory with an
                        arch-(archNo).xml file added representing
                        imagexList ]

    """
```

The XML generation technique uses the `xmlcreate.py` module; for more information, see the importation of this module in Section 4.2, “Imports” (p. 4).

We start by creating the document node. No `<!DOCTYPE ...>` will be attached to this XML file.

```

#-- 1 --
# [ doc := a new DOM Document object with root element
#       of type RNC_ARCHIVE_INDEX_N ]
doc = xc.Document ( RNC_ARCHIVE_INDEX_N )

```

Next we add child nodes to the root of this document, one for each element of `imagexList`. The actual generation of these child nodes is done by the `.writeNode()` method of the `Imagex` object; see Section 4.10, “`Imagex.writeNode(): Translate self to XML`” (p. 14).

```

#-- 2 --
# [ imagexList is a list of Imagex objects ->
#     doc.root := doc.root with nodes added representing
#             those objects ]
for imagex in imagexList:
    imagex.writeNode ( doc.root )

```

Finally, we write the resulting XML file to the index directory. The name of the index file is `INDEX_DIR + archDir`.

```

#-- 3 --
# [ index directory := index directory with an
#     (archDir+".xml") file added representing doc ]
fileName = "%s%s.xml" % (INDEX_DIR, archDir)
try:
    indexFile = open ( fileName, "w" )
except IOError, detail:
    print ( "*** Can't open index file '%s' for writing." %
           fileName )
    return
doc.write ( indexFile )
indexFile.close()

```

4.8. class Imagex: An object to describe one image

Each instance of this class holds the information about one image that we have indexed. The class knows how to add that information to a DOM tree for output as XML.

```

# - - - - - c l a s s   I m a g e x   - - - - -

class Imagex:
    """Represents information about one bird image.

    Exports:
    Imagex ( pathName ):
        [ (pathName is a string) ->
          if pathName names a readable, valid image file ->
            return a new Imagex object representing the image
          else ->
            raise IOError ]
    .pathName: [ as passed to constructor, read-only ]

```

```

.baseName:
    [ self.pathName, stripped of its directory part and
      extension ]
.image:      [ the image as an Image.Image object ]
.wide:       [ width in pixels as an integer ]
.high:       [ height in pixels as an integer ]
.writeNode ( parent ):
    [ parent is an xmlcreate.Element ->
      parent := parent with a new RNC_IMAGE_N node added
        representing self ]
"""

```

4.9. Imagex.__init__(): Constructor

Given the path name of an image file, we need to find the image size. Here's the constructor interface:

archx.py

```

# - - -   I m a g e x . _ _ i n i t _ _   - - -

def __init__ ( self, pathName ):
    """Constructor for Imagex.
    """
    #-- 1 --
    # [ self.pathName = pathName
    #   self.baseName = pathName, stripped of its directory
    #                   part and extension ]
    self.pathName = pathName
    discard, fileName = os.path.split ( pathName )
    self.baseName, discard = os.path.splitext ( fileName )

```

Python's `Image()` module does all the heavy lifting for us here. For documentation on this module, see *Python imaging library (PIL)*¹⁰.

This module's `Image.open()` method will raise an `IOError` exception in two different cases: if the file is inaccessible or nonexistent; and if the file does not represent one of the image formats supported by the `Image` module. In either case, we pass the exception back to our caller. If the file is readable and valid, we get back an `Image` object.

archx.py

```

#-- 2 --
# [ if pathName names a readable, valid image file ->
#   pic := an Image object representing that image
#   else ->
#     raise IOError ]
self.image = Image.open ( pathName )

```

The `.size` attribute of this object is a 2-tuple (width, height).

archx.py

```

#-- 3 --
# [ self.size gives (width,height) in pixels ->
#   self.wide := that width as mm

```

¹⁰ <http://www.nmt.edu/tcc/help/pubs/pil/>

```
# self.high := that height as mm ]
self.wide, self.high = self.image.size
```

4.10. `Imagex.writeNode()`: Translate self to XML

This method adds a representation of itself as an `<image>` element to the DOM tree.

archx.py

```
# - - - Imagex.writeNode - - -

def writeNode ( self, parent ):
    """Write an RNC_IMAGE_N node representing self.

    [ parent is an xmlcreate.Element object ->
      parent := parent with a new RNC_IMAGE_N node added
        representing self ]
    """
```

The `xc.Element` constructor accepts as an optional third argument a dictionary of attribute names and values. We first build up that dictionary, then `xc.Element` takes care of building the element and its attributes, and attaching it to the parent.

archx.py

```
attrs = { RNC_CAT_NO_A: self.baseName,
          RNC_WIDE_A:   str ( self.wide ),
          RNC_HIGH_A:  str ( self.high ) }
child = xc.Element ( parent, RNC_IMAGE_N, **attrs )
```

4.11. Epilogue

The last lines of the script execute the `main()` function, assuming that the script is being executed (not imported).

archx.py

```
#####
# Epilogue
#-----

if __name__ == "__main__":
    main()
```

5. `archindex.py`: Reader for archive index files

The `archindex.py` module reads one of the index files written by `archx.py`. For the interface, see Section 5.2, “class `ArchiveIndex`: Index of one archive” (p. 15).

5.1. Prologue

The code starts with the module’s documentation string and a few vital imports.

```

"""Reader module for files conforming to archx.rnc.
   For documentation, see:
   http://www.nmt.edu/~john/scans/slides/archx/
   """

```

The first `import`, enabling the use of Python generators, must precede all other `import` statements. Next we import `Parse` from the 4Suite XML package (see *Python and the XML Document Object Model (DOM) with 4Suite*¹¹) to read the XML file. Also included are two exceptions raised by the `Parse()` function, so we can catch them gracefully.

Note

This module has been ported to the more modern `lxml` library. See the copy in `tcc/p/cranefest/`, which should replace this module using the older 4Suite XML package.

```

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

from __future__ import generators
from Ft.Xml import Parse, ReaderException
from Ft.Lib import UriException
from rnc_archx import *

```

5.2. class ArchiveIndex: Index of one archive

Here is the interface to retrieve an archive index file. An `ArchiveIndex` is a container for `ArchiveImage` instances, each of which describes one archived image.

Typically you won't call the constructor directly; instead, use the static method `ArchiveIndex.readFile()` to read the file for you.

```

# - - - - - c l a s s   A r c h i v e I n d e x   - - - - -

class ArchiveIndex:
    """Represents one archive index file, conforming to archx.rnc.

    Exports:
    ArchiveIndex ( imageCatalog ):
        [ imageCatalog is a birdimages.ImageCatalog instance ->
          return a new, empty ArchiveIndex object ]
    .imageCatalog:      [ as passed to constructor ]
    .getArchImage ( catNo ):
        [ catNo is an image catalog number as a string ->
          if self has an entry for that catalog number ->
            return an ArchImage instance representing that entry
          else -> raise KeyError ]

```

¹¹ <http://www.nmt.edu/tcc/help/pubs/pyxml4>

```

.genArchImages():
    [ generate the ArchImage instances in self, in ascending
      order by catalog number ]
.addArchImage ( archImage ):
    [ archImage is an ArchImage instance ->
      self := self with archImage added ]
ArchiveIndex.readFile ( imageCatalog, fileName ):
    [ (imageCatalog is a birdimages.ImageCatalog instance) and
      (fileName is a string) ->
      if (fileName names an XML file valid against
          archx.rnc) and
          (catalog numbers in that file are all found in
           imageCatalog) ->
          return a new ArchiveIndex object containing ArchImage
          instances representing entries from fileName that
          do match entries in imageCatalog
      else -> raise IOError ]

```

The internal state of an ArchiveIndex instance consists of one dictionary:

archindex.py

```

State/Invariants:
    .__catNoMap:
        [ a dictionary whose values are the ArchiveIndex
          instances contained in self, and each key is the
          catalog number of that instance ]
    """

```

5.3. ArchiveIndex.__init__(): Constructor

There is little for this nominal constructor to do: just initialize the `.__catNoMap` dictionary.

archindex.py

```

# - - -   A r c h i v e I n d e x . _ _ i n i t _ _   - - -

def __init__ ( self ):
    """Constructor for ArchiveIndex.
    """
    self.__catNoMap = {}

```

5.4. ArchiveIndex.getArchImage(): Retrieve an archived image

Retrieves the ArchImage for the given catalog number, if any. If the catalog number is not in `self.__catNoMap`, this method will raise `KeyError`.

archindex.py

```

# - - -   A r c h i v e I n d e x . g e t A r c h I m a g e   - - -

def getArchImage ( self, catNo ):
    """Look up a catalog number.
    """
    return self.__catNoMap [ catNo ]

```

5.5. `ArchiveIndex.genArchImages()`: Generate contained image entries

Sorts the catalog numbers, then generates the corresponding `ArchImage` instances in that order.

`archindex.py`

```
# - - -   A r c h i v e I n d e x . g e n A r c h I m a g e s   - - -

def genArchImages ( self ):
    """Generate self's images.
    """
    catNoList = self.__catNoMap.keys()
    catNoList.sort()
    for catNo in catNoList:
        yield self.__catNoMap[catNo]
    raise StopIteration
```

5.6. `ArchiveIndex.addArchImage()`: Add one image

Add one `ArchImage` instance to `self`.

`archindex.py`

```
# - - -   A r c h i v e I n d e x . a d d A r c h I m a g e   - - -

def addArchImage ( self, archImage ):
    """Add one cataloged entry.
    """
    self.__catNoMap[archImage.original.catNo] = archImage
```

5.7. `ArchiveIndex.readFile()`: Instantiate from XML

This static method reads an XML file conforming to `archx.rnc` and returns its contents as an `ArchiveIndex` instance.

`archindex.py`

```
# - - -   A r c h i v e I n d e x . r e a d F i l e   - - -

# @staticmethod
def readFile ( imageCatalog, fileName ):
    """Read an XML file.
    """
```

We use the `Parse()` function to convert the XML file into a DOM tree.

`archindex.py`

```
#-- 1 --
# [ if fileName names a readable, well-formed XML file ->
#   doc := a DOM Document node representing that file
#   else -> raise IOError ]
try:
    doc = Parse ( fileName )
except UriException, detail:
    raise IOError, ( "No such file '%s': %s" %
                    (filename, detail) )
```

```

except ReaderException, detail:
    raise IOError, ( "File '%s' not well-formed: %s" %
                    (filename, detail) )

```

Next we build a node set of all the image nodes in the document, and also create an ArchiveIndex instance.

archindex.py

```

#-- 2 --
# [ xList := a node-set of all RNC_IMAGE_N nodes in doc
#   archx := a new, empty ArchiveIndex instance ]
xList = doc.documentElement.xpath ( '//%s' % RNC_IMAGE_N )
archx = ArchiveIndex()

```

Each valid node in xList will be converted to an ArchImage object and added to self.__catNoMap.

archindex.py

```

#-- 3 --
# [ if (all the nodes in xList are valid against
#   archx.rnc, and their catalog numbers are defined
#   in imageCatalog) ->
#   archx := archx with ArchImage instances added
#           representing valid nodes from xList
#   else -> raise IOError ]
for xNode in xList:
    #-- 3 body --
    # [ xNode is a DOM RNC_IMAGE_N Element node ->
    #   if xNode is not valid against archx.rnc ->
    #     raise IOError
    #   else if xNode's catalog number is found in
    #     self.imageCatalog ->
    #     archx := archx with an ArchImage instance
    #           added representing xNode ]

    #-- 3.1 --
    # [ (imageCatalog is a birdimages.ImageCatalog) and
    #   (xNode is a DOM RNC_IMAGE_N Element node) ->
    #   if (xNode is not valid against archx.rnc) or
    #     (xNode's catalog number is not in imageCatalog) ->
    #     raise IOError
    #   else ->
    #     archImage := an ArchImage instance
    #               representing that catalog number ]
    archImage = ArchImage.readNode ( imageCatalog, xNode )

    #-- 3.2 --
    # [ archx := archx with archImage added ]
    archx.addArchImage ( archImage )

```

Finally the accumulate catalog is returned to the caller.

archindex.py

```

#-- 4 --
return archx

readFile = staticmethod ( readFile )

```

5.8. class ArchImage: Archived catalog entry

Each instance of this class represents one image that is not only in the image catalog, but has also been measured for image size, and a thumbnail placed in the thumbnail directory. Most of the cataloging information is represented as an `Original` instance (as described in *An XML-based bird cataloging system*¹²), available as the `.original` attribute of an `ArchImage` instance.

Here is the class's interface, and its trivial constructor.

archindex.py

```
# - - - - - c l a s s   A r c h I m a g e   - - - - -

class ArchImage:
    """Represents the cataloging information for one archived image.

    Exports:
    ArchImage ( original, high, wide ):
        [ (original is a birdimages.Original instance) and
          (high is the image's height in pixels as an int) and
          (wide is the image's width in pixels as an int) ->
          return a new ArchImage object with those values ]
    .original:      [ as passed to constructor, read-only ]
    .high:          [ as passed to constructor, read-only ]
    .wide:          [ as passed to constructor, read-only ]
    ArchImage.readNode ( imageCatalog, xNode ):
        [ (imageCatalog is a birdimages.ImageCatalog instance) and
          (xNode is a DOM RNC_IMAGE_N Element ->
            if (xNode is not valid against archx.rnc) or
            (xNode's catalog number is not found in
            imageCatalog) ->
            raise IOError
          else ->
            return an ArchImage instance representing that
            catalog number ]
    """
    def __init__ ( self, original, high, wide ):
        """Constructor for ArchImage
        """
        self.original = original
        self.high     = high
        self.wide     = wide
```

5.9. ArchImage.readNode(): Convert an XML node

This static method converts an image node into an `ArchImage` instance, assuming that its catalog number is found in the dictionary.

archindex.py

```
# - - -   A r c h I m a g e . r e a d N o d e   - - -

#   @staticmethod
```

¹² <http://www.nmt.edu/~john/scans/slides/ims/>

```
def readNode ( imageCatalog, xNode ):
    """Convert an XML node to an ArchImage.
    """
```

First we pull out the catalog number, height, and width.

archindex.py

```
#-- 1 --
# [ catNo := xNode's RNC_CAT_NO_A attribute ]
catNo = xNode.getAttributeNS ( None, RNC_CAT_NO_A )

#-- 2 --
# [ if xNode has an RNC_HIGH_A attribute that is a valid
#   float in string form ->
#   high := that attribute as a float
#   else -> raise IOError ]
high = getIntAttr ( xNode, RNC_HIGH_A )

#-- 3 --
# [ if xNode has an RNC_WIDE_A attribute that is a valid
#   float in string form ->
#   wide := that attribute as a float
#   else -> raise IOError ]
wide = getIntAttr ( xNode, RNC_WIDE_A )
```

Translate the catalog number into an `Original` instance, or fail.

archindex.py

```
#-- 4 --
# [ if catNo matches a catalog number in imageCatalog ->
#   original := the corresponding Original from
#             imageCatalog
#   else -> raise IOError ]
original = imageCatalog.getOriginal ( catNo )

#-- 5 --
return ArchImage ( original, high, wide )

readNode = staticmethod ( readNode )
```

5.10. `getIntAttr()`: Retrieve an integer attribute value

This utility function handles the retrieval and conversion of an XML attribute that should contain an integer in string form.

archindex.py

```
# - - - g e t I n t A t t r - - -

def getIntAttr ( node, attrName ):
    """Convert an integer attribute value

    [ (node is a DOM Element node) and
      (attrName is an attribute name as a string) ->
        if node has an attribute named attrName and it
        contains a valid int in string form ->
```

```

        return that attribute as an int
    else -> raise IOError ]
    ""

#-- 1 --
# [ if node has an attribute named attrName ->
#     rawInt := that attribute's value
#     else -> raise IOError ]
rawInt = node.getAttributeNS ( None, attrName )
if not rawInt:
    raise IOError, ( "Missing %s attribute" % attrName )

#-- 2 --
# [ if rawInt is a valid int in string form ->
#     return int(rawInt)
#     else -> raise IOError ]
try:
    result = int ( rawInt )
    return result
except ValueError:
    raise IOError, ( "%s='%s': value not an int" %
                    (attrName, rawInt) )

```

6. Defect statistics

This program was written with Cleanroom intended functions. There was a gap of about three months at a point just before the writing of the `processFile()` function.

No self-verification or peer verification was done. Defects are counted from the first compilation. Defects are broken down into three categories: see Section 6.1, “Syntax errors” (p. 21); Section 6.2, “Strong typing errors” (p. 21); and Section 6.3, “Logic errors” (p. 22).

6.1. Syntax errors

Syntax errors caught by Python while scanning modules.

1. In `main()`, omitted the colon at the end of this line:

```
for archNo in sys.argv[1:]
```

6.2. Strong typing errors

These are errors that would have been caught at compile time in a more strongly typed language such as Java.

1. In `processFile()`, this line:

```
result.save ( thumbPath )
```

did not work, because `result` is an `Imagex` object. The desired method is on that object's attribute, `result.image`:

```
result.image.save ( thumbPath )
```

2. In `Imagex.writeNode()`, the original code to build the dictionary of attributes for the `RNC_IMAGE_N` node looked like this:

```
attrs = { RNC_CAT_NO_A: self.baseName,  
          RNC_WIDE_A:   self.wide,  
          RNC_HIGH_A:   self.high }
```

However, the DOM expects attribute values to be strings, and `self.wide` and `self.high` are floats. Some formatting fixes things right up:

```
attrs = { RNC_CAT_NO_A: self.baseName,  
          RNC_WIDE_A:   "%.1f" % self.wide,  
          RNC_HIGH_A:   "%.1f" % self.high }
```

The code above is no longer in the script; the width and height attributes have since been changed to integers.

3. When writing the `archindex.py` module, neglected to write the `ArchiveIndex.addArchImage()` method.

6.3. Logic errors

Ordinary logic errors.

1. Function `processFile()` declared a precondition that `pathName` cannot be the empty string. This precondition was added during design when I realized that the expression `baseName[0]` would fail with the empty string. I didn't think this was problem, because `os.listdir()` wouldn't produce an empty file name.

However, what is tested is the *base name*, which is the path name without its directory and extension. Unfortunately, a defect emerged in testing. The file name in question was the "hidden file" `bird-001/.xvpics`. Using `os.path.split()` on this string yields the tuple `('bird-001/', '.xvpics')`, and using `os.path.splitext()` on that produces `('', '.xvpics')`, and the expression `baseName[0]` fails with an `IndexError`.

The fix is to quietly return `None` whenever the `baseName` is empty, because hidden files cannot be bird images.

2. In `writeIndex()`, neglected to prepend `INDEX_DIR` to the name of the output file.