

PathInfo: A file information object in Python



John W. Shipman

2009-10-10 16:38

Abstract

Describes a class in the Python programming language that represents information about a file, as well as several useful scripts that use that class.

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: Why PathInfo? | 2 |
| 2. The interface to the PathInfo object | 3 |
| 3. <code>bigfiles.py</code> : Where are my biggest files? | 6 |
| 4. <code>oldfiles.py</code> : Sorting your files by modification date | 7 |
| 5. <code>softlinks.py</code> : Find soft links in a directory tree | 7 |
| 6. Source code for PathInfo | 8 |
| 6.1. Prologue | 8 |
| 6.2. Module constants | 8 |
| 6.3. <code>class PathInfo</code> | 9 |
| 6.4. <code>PathInfo.__init__()</code> : The class constructor | 9 |
| 6.5. <code>PathInfo.isFile()</code> : Is this an ordinary file? | 10 |
| 6.6. <code>PathInfo.isDir()</code> : Is this a directory? | 11 |
| 6.7. <code>PathInfo.isLink()</code> : Is this a soft link? | 11 |
| 6.8. <code>PathInfo.absPath()</code> : Absolute path | 11 |
| 6.9. <code>PathInfo.realPath()</code> : Actual absolute path | 12 |
| 6.10. <code>PathInfo.ownerCanRead()</code> , etc: Predicates for permission testing | 12 |
| 6.11. <code>PathInfo.modTime()</code> : Modification timestamp in human units | 13 |
| 6.12. <code>PathInfo.__str__()</code> : Convert to a string | 13 |
| 6.13. <code>PathInfo.__fileType()</code> : Get the file type code | 13 |
| 6.14. <code>PathInfo.__permFlags()</code> : Format all the permissions | 14 |
| 6.15. <code>PathInfo.__rwx()</code> : Format three permission bits | 15 |
| 6.16. <code>PathInfo.__dasher()</code> : Format a permission bit | 15 |
| 6.17. <code>PathInfo.__cmp__()</code> : Define the comparison operator on PathInfo objects | 15 |
| 7. Source code for <code>bigfiles.py</code> | 16 |
| 7.1. <code>bigfiles.py</code> : Code prologue | 17 |
| 7.2. <code>bigfiles.py</code> : The main program | 17 |

¹ <http://www.nmt.edu/tcc/help/lang/python/examples/pathinfo/>

² <http://www.nmt.edu/tcc/help/lang/python/examples/pathinfo/pathinfo.pdf>

| | |
|---|----|
| 7.3. <code>report()</code> : Generate one directory tree's report | 18 |
| 7.4. <code>class BigInfo</code> : The <code>PathInfo</code> subclass | 19 |
| 7.5. <code>BigInfo.__init__()</code> : Constructor | 19 |
| 7.6. <code>BigInfo.__cmp__()</code> : The comparator method | 20 |
| 7.7. <code>BigInfo.__str__()</code> : String conversion method | 21 |
| 7.8. <code>class BigReport</code> : The class for the whole application | 22 |
| 7.9. <code>BigReport.__init__()</code> : Constructor | 22 |
| 7.10. <code>BigReport.__visitor()</code> : Visitor function for <code>os.path.walk()</code> | 23 |
| 7.11. <code>BigReport.genFiles()</code> : Generate the report | 25 |
| 7.12. Epilogue | 25 |
| 8. Source code for <code>oldfiles.py</code> | 25 |
| 8.1. <code>oldfiles.py</code> : Code prologue | 26 |
| 8.2. <code>oldfiles.py</code> : The main program | 26 |
| 8.3. <code>report()</code> : Generate one directory tree's report | 27 |
| 8.4. <code>class OldInfo</code> : The <code>PathInfo</code> subclass | 27 |
| 8.5. <code>OldInfo.__init__()</code> | 28 |
| 8.6. <code>OldInfo.__cmp__()</code> : The comparator method | 28 |
| 8.7. <code>OldInfo.__str__()</code> : String conversion method | 29 |
| 8.8. <code>class OldReport</code> : The application class | 29 |
| 8.9. <code>OldReport.__init__()</code> : Constructor | 30 |
| 8.10. <code>OldReport.__visitor()</code> : Visitor function for <code>os.path.walk()</code> | 30 |
| 8.11. <code>OldReport.genFiles()</code> : Generate the report | 31 |
| 8.12. Epilogue | 32 |
| 9. Source code for <code>softlinks.py</code> | 32 |
| 9.1. <code>softlinks.py</code> : Code prologue | 32 |
| 9.2. <code>main()</code> : Main program | 33 |
| 9.3. <code>report()</code> : Generate one tree's report | 33 |
| 9.4. <code>class LinkReport</code> : Link report container | 34 |
| 9.5. <code>LinkReport.__init__()</code> : Constructor | 35 |
| 9.6. <code>LinkReport.__visitor()</code> : Visitor function for <code>os.path.walk()</code> | 35 |
| 9.7. <code>LinkReport.genLinks()</code> : Generate links | 36 |
| 9.8. <code>class LinkInfo</code> : The <code>PathInfo</code> subclass | 36 |
| 9.9. <code>LinkInfo.__init__()</code> : Constructor | 37 |
| 9.10. <code>LinkInfo.__str__()</code> : Convert to a string | 37 |
| 9.11. Epilogue | 38 |

1. Introduction: Why PathInfo?

This document describes a small Python class called `PathInfo`. This class was written for two reasons:

- Each instance of a `PathInfo` object represents a snapshot of the state of a file, directory, or other such object, with a number of methods to help you look at aspects of the file such as its size and permissions.
- It is a good example of a small Python class for use in introducing object-oriented programming to new Python programmers.

Also, we present some small scripts that use the `PathInfo` object for practical, everyday tasks:

- When you have a lot of disk space tied up and want to find some large files you can offload or delete, you want to see a list of your largest files in descending order by size. See Section 3, “`bigfiles.py`: Where are my biggest files?” (p. 6).

- Sometimes you want to look at files according to when they were last modified. For example, you might want to find the files you've used least recently to help you weed out unused files. Or you might be trying to find some files inside a large file structure, and you don't know where they are, but you do remember *when* you were working on them. See Section 4, “`oldfiles.py`: Sorting your files by modification date” (p. 7).

This project exhibits a number of useful software technologies. In addition to object-oriented programming:

- This document includes all the Python source code for the `PathInfo` object and the example scripts that use it. For more on this practice, called *literate programming*, see *A source extractor for lightweight literate programming*³.
- The code was developed using the Cleanroom or zero-defect style; for the author's practice and conventions, see *The cleanroom software development methodology*⁴.

Source files contained in this package include:

- `pathinfo.py`⁵: The module containing the `PathInfo` class.
- `bigfiles.py`⁶: The script for finding large files.
- `oldfiles.py`⁷: The script for sorting files by modification time.

2. The interface to the `PathInfo` object

To use the `PathInfo` object, import the `pathinfo.py` module like this:

```
from pathinfo import *
```

Then, to take a snapshot of the state of a file or directory, instantiate a `PathInfo` object like this:

```
info = PathInfo ( path )
```

where *path* is the path name (absolute or relative) of the file of interest.

For example, suppose you want to take a snapshot of the state of the file `/etc/crontab`:

```
cronInfo = PathInfo ( "/etc/crontab" )
```

If the given path does not exist or is not accessible, the `PathInfo` constructor will raise an `OSError` exception.

Here are the attributes and methods of a `PathInfo` object.

.path

The pathname that was passed to the constructor. This is a read-only attribute: do not modify it in place.

.size

The size of the file in bytes.

³ <http://www.nmt.edu/tcc/help/lang/python/examples/litsource>

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

⁵ <http://www.nmt.edu/tcc/help/lang/python/examples/pathinfo/pathinfo.py>

⁶ <http://www.nmt.edu/tcc/help/lang/python/examples/pathinfo/bigfiles.py>

⁷ <http://www.nmt.edu/tcc/help/lang/python/examples/pathinfo/oldfiles.py>

.createEpoch

The creation time of the file as an *epoch time*. For more information on epoch time, refer to the Python documentation for the `time` module⁸.

.modEpoch

The time when the file was last modified, as an epoch time.

.isFile()

A predicate that tests whether the `PathInfo` object describes an ordinary file.

.isDir()

A predicate that tests whether the object describes a directory.

.isLink()

A predicate that tests whether the object describes a soft (symbolic) link. For more information on soft links, see *How to make a file appear in multiple directories*⁹.

.absPath()

Returns the absolute path name to this file. If there are any soft links in the path, their names are not replaced with the real paths.

.realPath()

Returns the absolute path name to this file. If there are any soft links in the path, their names *are* replaced with the real paths.

.ownerCanRead()

A *predicate* that tests whether the file has read permission for the owner. A predicate is a method that returns a Boolean result: in this case, it returns a `True` result if the file has owner read permission, or a `False` result if not.

For more information on permissions, see *Controlling access to your files*¹⁰.

.ownerCanWrite()

A predicate that tests whether the file's owner has write permission.

.ownerCanExec()

A predicate that tests whether the file's owner has execute permission.

.groupCanRead()

A predicate that tests whether the file has group read permission.

.groupCanWrite()

Predicate to test for the group write permission.

.groupCanExec()

Predicate to test for the group execute permission.

.worldCanRead()

Predicate to test the world read permission.

.worldCanWrite()

Predicate to test the world write permission.

.worldCanExec()

Predicate to test the world execute permission.

⁸ <http://docs.python.org/lib/module-time.html>

⁹ <http://www.nmt.edu/tcc/help/unix/ln.html>

¹⁰ <http://www.nmt.edu/tcc/help/unix/fileaccess.html>

.modTime()

This method formats the modification timestamp of the path as a string in a consistent format: "yyyy-mm-dd hh:mm:ss", where *yyyy* is the year, *mm* is the month number, and so on. The advantage of using month numbers instead of month names is that this string can be used as a sort key and will order records in chronological order.

We use local time because it is more user-friendly than Universal Time.

.__str__()

The class contains a special method named `__str__()`. This method is called whenever an instance needs to be converted into a string, as for example if you use a `PathInfo` object in a `print` statement, or when you call the `str()` function on a `PathInfo` instance.

The value returned by this function is intended to supply general information about the file, such as you might get from the Unix `ls -l` command. The author dislikes the output of `ls -l` because the data format is irregular. Here are two example lines to illustrate this:

```
-rw-r--r--    1 john    tcc          1235 Dec 30  2003 vdrybean
-rw-----    1 john    tcc          15738 Jul  4 08:32 x
```

File `vdrybean` was last modified in 2003, but we don't know what time that day. File `x`, on the other hand, was modified last July 4, and we know when, but note that the proper interpretation of this line requires the reader to know *when* the listing was made. Also, it is difficult to sort on this date format.

Therefore, the string returned by this method has the format `drwxrwxrwx yyyy-mm-dd hh:mm:ss size p`, where:

- `drwxrwxrwx` shows the mode bits in the same format as the Unix `ls -l` command. The first character is "-" for an ordinary file, "d" for a directory, and "l" for a soft link. The next three letters show the read (r), write (w), and execute (x) permissions for the file owner. Next come the same three permissions for group members, then the world permissions.
- The date format displays the modification time in the format described above under the `.modTime()` method.
- The `size` is the file size in bytes. By default we allow a generous 8 digits in this field, enough for files up to 99,999,999 bytes, but the field will expand if necessary.
- The pathname `p` follows.

.__cmp__()

Python allows classes to define a `.__cmp__()` method that is used whenever two objects of that class are compared. It affects the use of relational operators like "`>=`", calls to the `cmp()` function, and use of the list type's `.sort()` method, among others.

This method takes two arguments, the usual `self` as its first argument, and a second argument that provides the other object to be compared. It should return a value using the same conventions as the built-in Python comparison function, `cmp()`: a negative number if `self` comes before `other`, a positive number if `self` should follow `other`, and zero if they are equal.

In this particular case, the `.__cmp__()` method orders `PathInfo` objects by their `.path` attributes.

.status

The `status tuple` for this file. This is a low-level data structure from which much of the other information is derived. It is exported to the caller as a safety valve for features that the `PathInfo` object

does not support, such as access to the time when a file was last read. For more information on the status tuple, see the Python documentation for the file-related operations of the `os` module¹¹.

.mode

The “mode bits” from the status tuple. This integer includes permissions and a few other fields. As with the `.status` attribute, this is exported in case the supplied methods are not sufficient for some applications.

Here's a brief conversational example showing use of the `PathInfo` object. First, here's the output from the “`ls -l`” command for a file called `re`:

```
-rw-r--r--  1 john  1659          30 Jul 20 11:03 re
```

Using Python's conversational mode, let's look at these same attributes through a `PathInfo` object.

```
>>> import pathinfo
>>> re=pathinfo.PathInfo('re')
>>> re.path
're'
>>> re.size
30L
>>> re.isFile()
1
>>> re.isDir()
0
>>> re.isLink()
0
>>> re.ownerCanRead()
256
>>> re.ownerCanWrite()
128
>>> re.ownerCanExec()
0
>>> re.worldCanRead()
4
>>> re.worldCanWrite()
0
>>> re.worldCanExec()
0
>>> re.modTime()
'2005-07-20 11:03:02'
>>> print re
-rw-r--r-- 2005-07-20 11:03:02          30 re
>>>
```

3. bigfiles.py: Where are my biggest files?

The first application for the `PathInfo` object is in the `bigfiles.py` script, which shows you all the files in some directory or directories, sorted in descending order by size.

Here is the general form for the command line invocation of this script:

¹¹ <http://docs.python.org/lib/os-file-dir.html>

```
bigfiles.py [dir ...]
```

The script takes as arguments zero or more pathnames of directories. For each argument, it produces a report listing all the ordinary files (not directories or soft links) in that directory tree. If you don't give it any directory names, it writes a report on ".", the current working directory.

Each report starts off with a line showing the actual absolute path to the selected directory.

Each line of the report has this format:

```
yyyy-mm-dd hh:mm:ss size pathname
```

where:

- *yyyy-mm-dd hh:mm:ss* is the date and time when the file was last modified.
- *size* is the size of the file in bytes.
- *pathname* is the path name of the file, *relative to the starting directory* that was named on the command line.

So, for example, if you used this command:

```
bigfiles /u/guykay
```

then if a pathname appears in the report as "bin/sarantium/crispin", that file's actual pathname is "/u/guykay/bin/sarantium/crispin".

4. oldfiles.py: Sorting your files by modification date

This script produces a report in the same format as does `bigfiles.py`. The only difference is that the files are presented in reverse chronological order, that is, in descending order by modification time.

The command line arguments are the same: a list of directories. If no arguments are given, you get a report of the current working directory.

5. softlinks.py: Find soft links in a directory tree

The `softlinks.py` script will find all the Unix-style soft links in a given list of directory trees. For each soft link, it shows the target path to which the link points, and it will also flag *broken links*, that is, soft links whose target is a path that does not exist.

The command line arguments are the path names of the directory trees you want to search. If you provide no arguments, the search begins in the current working directory ".".

```
softlinks.py [dir ...]
```

Each soft link results in two lines of output. Here is the general form, when the soft link points to an existing file:

```
linkPath ->  
targetPath
```

where *linkPath* is the path to the soft link, and the *targetPath* is the absolute path name where the link points.

If the link is broken, a string of pound signs is added so that broken links stand out:

```
linkPath ->
##### targetPath
```

6. Source code for PathInfo

Here is the source code for the `pathinfo.py` module containing the `PathInfo` class.

6.1. Prologue

The code for the `pathinfo.py` module starts with the conventional Python documentation string.

```
"""pathinfo.py: Object to represent a snapshot of a file's status.
For documentation in "literate programming" style, see:
http://www.nmt.edu/help/lang/python/examples/pathinfo/
"""
```

`pathinfo.py`

Next comes the importation of the standard Python modules we use:

- The `os` module supports Posix file systems and other operating system functions.
- The `stat` module supplies additional declarations needed for interpreting the information that comes out of the `os.stat()` and `os.lstat()` functions.
- We use the `time` module for translating and formatting file timestamps. In Python 2.3 and later, the `datetime` module is now recommended, but we'll use the older version out of consideration for those who have older Python installs.

```
#=====
# Imports
#-----
import os, stat, time
```

`pathinfo.py`

6.2. Module constants

We define one manifest constant named `TIME_FORMAT` that controls the formatting of time tuples into textual form. This is used in the `PathInfo.__str__()` method. It is exported by this module so that other classes that inherit from `PathInfo` can format other timestamps in the same way. The output is in the form "yyyy-mm-dd hh:mm:ss".

```
#=====
# Manifest constants
#-----
TIME_FORMAT = "%Y-%m-%d %H:%M:%S"
```

`pathinfo.py`

6.3. class PathInfo

The rest of the code in this module is inside the `PathInfo` class. We state here in the class's documentation string the class *invariants*: conditions that must always be true once the constructor has completed. For more on invariants, see the author's notes on cleanroom verification of objects¹².

pathinfo.py

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

# - - - - - c l a s s   P a t h I n f o   - - - - -

class PathInfo:
    """Represents a snapshot of one file's status.

    Class invariants:
    .path:
        [ the pathname passed to the class constructor ]
    .size:
        [ self.path's size in bytes as an integer ]
    .createEpoch:
        [ the epoch time when self.path was created ]
    .modEpoch:
        [ the epoch time when self.path was last modified ]
    .mode:
        [ the mode bits for self.path ]
    """
```

6.4. PathInfo.__init__(): The class constructor

This method is the one actually called when you use the class constructor `PathInfo()`. The single argument is the pathname of the desired file or directory.

pathinfo.py

```
# - - -   P a t h I n f o .   _ _ i n i t _ _   - - -

def __init__( self, path ):
    """Constructor for the PathInfo class.

    [ path is a string ->
      if path names an inode whose status is readable ->
        return a new PathInfo containing that status
      else -> raise OSError ]
    """
```

The job of a constructor is to assemble a new instance of the class. In Python classes, all methods of a class have an invisible first argument `self`: this is the instance on which the methods operate. The instance can be thought of as a *namespace*—that is, a set of names and their corresponding values. When the constructor starts, `self` is a namespace containing all the names of methods in the class (and the

¹² <http://infohost.nmt.edu/~shipman/soft/clean/ooverf.html>

class variables, of any), but no other names. The constructor then customizes the instance by adding or modifying the names in the namespace `self`.

The first order of business is to save the argument inside the instance's namespace, symbolized by `self`.

pathinfo.py

```
#-- 1 --
self.path = path
```

Next we ask the operating system for the status of that pathname.

There are two different functions in the `os` module for getting the status information about a path: `os.stat()` and `os.lstat()`. They work the same except when the path points to a soft link. In that case, `os.stat()` retrieves data about the path pointed to by the soft link, while `os.lstat()` retrieves data about the soft link itself. For our purposes, we want the latter. Both these methods raise an `OSError` exception if the given path is nonexistent or inaccessible.

pathinfo.py

```
#-- 2 --
# [ if path names an existing, accessible file path ->
#   self.status := the status tuple for path
#   else -> raise OSError ]
self.status = os.lstat ( path )
```

Then we pull out the items from the status tuple that we export: the file size, the creation and modification timestamps, and the mode bits. The constants that start with “`ST_...`” are the indices of elements of the status tuple, and come from the `stat` module.

pathinfo.py

```
#-- 3 --
# [ self.status is a status tuple ->
#   self.size := size from self.status
#   self.createEpoch := creation epoch time from
#                       self.status
#   self.modEpoch := modification epoch time
#                   from self.status
#   self.mode := mode bits from self.status ]
self.size = self.status[stat.ST_SIZE]
self.createEpoch = self.status[stat.ST_CTIME]
self.modEpoch = self.status[stat.ST_MTIME]
self.mode = self.status[stat.ST_MODE]
```

At this point we are done, because we have established the five stated invariants on the attributes `.path`, `.size`, `.createEpoch`, `.modEpoch`, and `.mode`.

Because this is a constructor (that is, because its name is `__init__()`), we don't have to return a value explicitly. The constructor's first argument, `self`, is the instance we have constructed, and it is returned to the caller of the `PathInfo()` constructor.

6.5. `PathInfo.isFile()`: Is this an ordinary file?

The standard Python `stat` module contains a function called `S_ISREG()` that takes the mode bits as an argument and returns true if the mode bits describe a regular file, false otherwise.

pathinfo.py

```
# - - - PathInfo.isFile - - -
```

```

def isFile ( self ):
    """Predicate to test whether this path is an ordinary file.

    [ if self represents an ordinary file ->
      return a true value
      else ->
        return a false value ]
    """
    return stat.S_ISREG ( self.mode )

```

6.6. PathInfo.isDir(): Is this a directory?

The `stat` module has a predicate named `S_ISDIR()` that tests the mode bits to see if they describe a directory.

pathinfo.py

```

# - - -   P a t h I n f o . i s D i r   - - -

def isDir ( self ):
    """Predicate to test whether this path is a directory.

    [ if self represents a directory ->
      return a true value
      else ->
        return a false value ]
    """
    return stat.S_ISDIR ( self.mode )

```

6.7. PathInfo.isLink(): Is this a soft link?

The `stat` module has a predicate for testing for soft links: `S_ISLNK()`.

pathinfo.py

```

# - - -   P a t h I n f o . i s L i n k   - - -

def isLink ( self ):
    """Predicate to test whether this path is a soft link.

    [ if self represents a soft link ->
      return a true value
      else ->
        return a false value ]
    """
    return stat.S_ISLNK ( self.mode )

```

6.8. PathInfo.absPath(): Absolute path

This method returns the absolute path name equivalent to `self.path`. It uses the standard Python function `os.path.abspath()`.

pathinfo.py

```

# - - -   P a t h I n f o . a b s P a t h   - - -

```

```
def absPath ( self ):
    """Return self's absolute path name."""
    return os.path.abspath ( self.path )
```

6.9. PathInfo.realPath(): Actual absolute path

This method returns the absolute path name equivalent to `self.path`. Unlike the `.absPath()` method, any soft links that may occur within the path are resolved and replaced with the paths to which the links refer. It uses the standard Python function `os.path.realpath()`.

pathinfo.py

```
# - - - P a t h I n f o . r e a l P a t h - - -

def realPath ( self ):
    """Return self's absolute path name, with links resolved."""
    return os.path.realpath ( self.path )
```

6.10. PathInfo.ownerCanRead(), etc: Predicates for permission testing

All nine of the routines that test read, write, and execute permissions are structurally identical. In each case, one of the mode bits is 1 if the path has that permission, or 0 otherwise.

To test this bit while ignoring the other bits, we use a *mask* that has a 1 bit in that position and 0 bits elsewhere. Applying a Boolean “and” operation on the mask and the mode bits gives us a word that is all zeroes (false) if the relevant bit is 0, or a word that is not zero (true) if the relevant bit is not zero.

The first method illustrates the pattern. The mask for the owner read permission bit comes from the `stat` module, and is called `S_IRUSR`.

pathinfo.py

```
# - - - P a t h I n f o . { o w n e r } C a n { R e a d } - - -
#                               { g r o u p }       { W r i t e }
#                               { w o r l d }       { E x e c }

def ownerCanRead ( self ):
    return self.mode & stat.S_IRUSR
```

The remaining routines are identical except for the names of the masks they use from the `stat` module.

pathinfo.py

```
def ownerCanWrite ( self ):
    return self.mode & stat.S_IWUSR

def ownerCanExec ( self ):
    return self.mode & stat.S_IXUSR

def groupCanRead ( self ):
    return self.mode & stat.S_IRGRP

def groupCanWrite ( self ):
    return self.mode & stat.S_IWGRP

def groupCanExec ( self ):
    return self.mode & stat.S_IXGRP
```

```

def worldCanRead ( self ):
    return self.mode & stat.S_IROTH

def worldCanWrite ( self ):
    return self.mode & stat.S_IWOTH

def worldCanExec ( self ):
    return self.mode & stat.S_IXOTH

```

6.11. PathInfo.modTime(): Modification timestamp in human units

This method translates `self.modEpoch` to our standard date and time format.

First we convert the epoch time to a local time tuple, then we format it using the time formatting method of the `time` module.

pathinfo.py

```

# - - -   m o d T i m e   - - -

def modTime ( self ):
    """Format the modification time as yyyy-mm-dd hh:mm:ss."""
    return time.strftime ( TIME_FORMAT,
        time.localtime ( self.modEpoch ) )

```

6.12. PathInfo.__str__(): Convert to a string

For the format of the string returned by this method, see Section 2, "The interface to the `PathInfo` object" (p. 3).

We call internal methods to develop the file type, file permissions, and file modification timestamp. The assembly of the pieces, and the formatting of the size, is handled by the usual Python string format operator.

pathinfo.py

```

# - - -   P a t h I n f o . _ _ s t r _ _   - - -

def __str__ ( self ):
    """Convert self to a string."""
    return ( "%s%s %s %8d %s" %
        (self.__fileType(), self.__permFlags(),
        self.modTime(), self.size, self.path) )

```

6.13. PathInfo.__fileType(): Get the file type code

This method derives the one-letter file type code: `-` for a regular file, `d` for a directory, or `l` for a link. Because it is a private method of the class, its name starts with two underscores (`__`) so it is not visible to code that imports this class.

pathinfo.py

```

# - - -   P a t h I n f o . _ _ f i l e T y p e   - - -

def __fileType ( self ):
    """Return the file type code.

```

```

    [ if self is a regular file ->
      return "-"
      if self is a directory ->
        return "d"
      if self is a soft link ->
        return "l" ]
    """

```

Just as a defensive programming measure, we return "?" if for some reason the path is neither a file, a directory, or a soft link. This can happen for Unix device files, but those are beyond the intended audience of the `pathinfo.py` module.

pathinfo.py

```

    if self.isLink():
        return "l"
    elif self.isDir():
        return "d"
    elif self.isFile():
        return "-"
    else:
        return "?"

```

6.14. PathInfo.__permFlags(): Format all the permissions

This method formats the permission bits in `self.mode` using the time-honored format of the Unix "ls" command.

pathinfo.py

```

# - - - P a t h I n f o . _ _ p e r m F l a g s - - -

def __permFlags ( self ):
    """Format self.mode's permissions as 'rwxrwxrwx'."""
    """

```

Each set of three permission bits is formatted in the same way, so we call method `self.__rwx()` to format them. This method takes three arguments. The first argument is 0 if there is no read permission, nonzero otherwise. The second and third arguments are the write and execute permission with the same convention.

To get the values for each permission, we use a Boolean "and" (&) operator on `self.mode` and the mask values from the `stat` module to discard all but the bit of interest. Mask `stat.S_IRUSR` has a one bit in the position of the owner read permission of the mode word, and zeroes in the other positions. Mask `stat.S_IWUSR` is a mask for the owner write permission, and so on.

pathinfo.py

```

    return ( "%s%s%s" %
            (self.__rwx ( self.mode & stat.S_IRUSR,
                          self.mode & stat.S_IWUSR,
                          self.mode & stat.S_IXUSR ),
              self.__rwx ( self.mode & stat.S_IRGRP,
                          self.mode & stat.S_IWGRP,
                          self.mode & stat.S_IXGRP ),
              self.__rwx ( self.mode & stat.S_IROTH,
                          self.mode & stat.S_IWOTH,
                          self.mode & stat.S_IXOTH ) ) )

```

6.15. PathInfo.__rwx(): Format three permission bits

This little method takes three permission values and returns a three-character string formatted in the “ls -l” convention. The read permission is formatted as “r” if true, “-” if false. Similarly, the write permission is formatted as “w” or “-”, and the execute permission as “x” or “-”.

Each argument is nonzero if the permission is set, zero if it is not set.

pathinfo.py

```
# - - -   P a t h I n f o . _ _ r w x   - - -  
  
def __rwx ( self, r, w, x ):  
    """Format three permission bits.  
  
    [ r, w, and x are Boolean values indicating read,  
      write and execute permissions ->  
      return a three-character string displaying those  
      permissions as "ls -l" displays them ]  
    """
```

The `__dasher()` method handles generation of either a letter or a dash depending on the permission value.

pathinfo.py

```
        return ( "%S%S%S" %  
                (self.__dasher ( r, "r" ),  
                  self.__dasher ( w, "w" ),  
                  self.__dasher ( x, "x" ) ) )
```

6.16. PathInfo.__dasher(): Format a permission bit

Formats a permission bit using the “ls -l” convention. The `bit` argument is true if the permission is granted, false otherwise. The `flag` argument is returned if the permission is true, otherwise the method returns “-”.

pathinfo.py

```
# - - -   P a t h I n f o . _ _ d a s h e r   - - -  
  
def __dasher ( self, bit, flag ):  
    """Format a permission bit as in ls -l.  
  
    [ (bit is a Boolean value) and (flag is a string) ->  
      if bit is true ->  
        return flag  
      else -> return "-" ]  
    """  
    if bit: return flag  
    else:   return "-"
```

6.17. PathInfo.__cmp__(): Define the comparison operator on PathInfo objects

We want PathInfo objects to sort by their `.path` attributes, that is, in ascending order by pathname.

Because these pathnames are simple strings, and because Python's built-in `cmp()` function can compare strings, all we have to do here is call `cmp()` on the pathnames and return its result as our result.

pathinfo.py

```
# - - - P a t h I n f o . _ _ c m p _ _ - - -  
  
def __cmp__( self, other ):  
    """Comparison operator for PathInfo objects.  
  
    [ other is a PathInfo object ->  
      if self.path < other.path ->  
        return a negative number  
      else if self.path > other.path ->  
        return a positive number  
      else ->  
        return 0 ]  
    """  
    return cmp ( self.path, other.path )
```

7. Source code for `bigfiles.py`

The script starts out by making up a list of directory trees to be visited. If there are any command line arguments, these arguments make up the list of directories. If there are no arguments, we default to a list containing one entry, ".".

The following steps are done for each directory in the list:

1. Look at every file in and under that directory. Use `PathInfo` to take a snapshot of that file, and build a list of these `PathInfo` instances. This process of "walking the directory tree" is easy because the `os.path` module has a function called `walk()` that handles the process of visiting every directory in the tree. For documentation on this function, see the *Python Library Reference*¹³.
2. Sort this list in descending order by the size attribute.

Python makes it easy to sort a list: all list objects have a `.sort()` method that sorts the list in place. But how do we get a list of `PathInfo` objects to sort in descending order by size? The `.sort()` method can take as an optional argument a function that compares two objects. However, a more Pythonic way to do it is to define a new class that inherits from the `PathInfo` class called `BigInfo`. In that class we define a special method named `__cmp__()` that tells Python how to order two objects of that class.

3. Print a heading for this section of the report, then go through the sorted list and print one line for each `PathInfo` instance in that list.

One of the goals of object-oriented programming is to minimize, if not eliminate, the use of global variables. An earlier version of this program used a global variable to hold the list of `PathInfo` objects. A better way is to define a class that holds this list. The methods in the class have access to the list, but no code outside the class needs such access. This is in accord with the generally accepted software design principle of "information hiding": we will feed the class constructor the name of a directory, and it will return an object that has everything we need to generate the final report.

We'll call this class `BigReport`, because it represents a report on big files. With this design, the overall program flow for each directory becomes:

¹³ <http://docs.python.org/lib/module-os.path.html>

1. Instantiate a `BigReport` object. Pass the name of the directory to its constructor.
2. The `BigReport` object has a method named `.genFiles()` that generates the lines of the report. (Generators are a relatively new feature of Python, since version 2.2. See the Python language reference¹⁴ section on generators.)

7.1. `bigfiles.py`: Code prologue

The actual code for the `bigfiles.py` script starts with the Linux “pound bang line” that makes the script self-executing.

```
bigfiles.py
```

```
#!/usr/bin/env python
#=====
# bigfiles.py:  Script to show files in descending order by size.
#   For documentation in "literate programming" style, see:
#     http://www.nmt.edu/help/lang/python/examples/pathinfo/
#-----

SCRIPT_NAME      = "bigfiles.py"
EXTERNAL_VERSION = "1.1"
```

Next we need to import a few Python modules: `sys` for command line arguments and standard streams; `os` for numerous file- and directory-related functions; and of course `pathinfo.py`.

```
bigfiles.py
```

```
#=====
# Imports
#-----
import sys, os
import pathinfo
```

7.2. `bigfiles.py`: The main program

The first step is to put together a list of the desired directories. If none are given on the command line, we create a list containing just `."` for the current directory.

```
bigfiles.py
```

```
# - - - - -   m a i n   - - - - -

def main():
    """Main program."""

    print "=== %s %s ===" % (SCRIPT_NAME, EXTERNAL_VERSION)

    #-- 1 --
    # [ if sys.argv[1:] is empty ->
    #   dirList := [ "." ]
    # else ->
    #   dirList := sys.argv[1:] ]
    dirList = sys.argv[1:]
    if len(dirList) == 0:
        dirList = [ "." ]
```

¹⁴ <http://docs.python.org/ref/yield.html>

Next we go through the elements of `dirList`, generating a report for each one.

`bigfiles.py`

```
#-- 2 --
# [ sys.stdout += reports listing files below each
#     directory named in dirList, with files in descending
#     order by size ]
for dirName in dirList:
    #-- 2 body --
    # [ dirName is a string ->
    #     sys.stdout += a report listing the files below
    #     directory (dirName), with files in descending
    #     order by size ]
    report ( dirName )
```

7.3. `report()`: Generate one directory tree's report

The `report()` function generates the portion of the report for one directory subtree. The path name to the subtree is its argument.

`bigfiles.py`

```
# - - -   r e p o r t   - - -

def report ( dirName ):
    """Generate the report for one directory subtree.

    [ dirName is a string ->
      sys.stdout += a report listing the files below
      directory (dirName), with files in descending
      order by size ]
    """
```

This function has only three steps: write a report heading; instantiate the `BigReport` object containing the report data; and call that object's `.genFiles()` method to generate the lines of the report.

Each report starts with a line showing the name of the subtree's starting directory. This uses Python's `os.path.realpath()` function, which resolves soft links and relative path names to the actual absolute path name.

We also set `basePath` to the absolute path name corresponding to `dirName`. This is necessary to the `BigReport` object so that it can display each file's path name relative to that base directory. For this, we use the `os.path.abspath()` function, which does not replace soft links with their real locations.

`bigfiles.py`

```
#-- 1 -
# [ basePath := dirName's absolute path name
#     sys.stdout += report heading showing dirName's real
#     absolute path ]
basePath = os.path.abspath ( dirName )
print "\n   === %s ===" % os.path.realpath ( dirName )

#-- 2 --
# [ bigReport := a BigReport object describing all the
#     accessible files in directory tree (dirName) ]
bigReport = BigReport ( basePath )
```

```

#-- 3 --
# [ bigReport is a BigReport object ->
#   sys.stdout += lines describing files in bigReport
#   in descending order by size ]
for bigInfo in bigReport.genFiles():
    print bigInfo

```

7.4. class **BigInfo**: The **PathInfo** subclass

In order to make the file snapshots sort in descending order by size, we could just define a `.__cmp__()` method in the `PathInfo` class. However, the `bigfiles.py` script and the `oldfiles.py` script need different sorting behavior: the former sorts by size, while the latter sorts by modification timestamp.

So each of these scripts defines a new class, inheriting from `PathInfo`, that defines a `.__cmp__()` method that makes the objects sort correctly for that application.

So that a `BigInfo` instance can display the path name relative to the report's starting directory, its constructor requires an additional argument named `basePath`, the starting directory's absolute path.

Here's the beginning of the class declaration. Note that the class name is followed by the parent class name in parentheses.

```

                                                                    bigfiles.py
#=====
# Functions and classes
#-----

# - - - - - c l a s s   B i g I n f o   - - - - -

class BigInfo(pathinfo.PathInfo):
    """Represents information about one file; sorts by size.

    Exports:
        BigInfo ( path, basePath ):
            [ (path is the path name to a file) and
              (basePath is the path name of some directory above
                path) ->
              return a new BigInfo instance with those values ]
        .__cmp__ ( self, other ):
            [ other is a BigInfo instance ->
              return cmp ( other.size, self.size ) ]
        .__str__ ( self ):
            [ return a string describing self's modification time,
              its size, and its path name relative to basePath ]

    State/Invariants:
        .__basePath: [ as passed to constructor, read-only ]
    """

```

7.5. **BigInfo**.`__init__()`: Constructor

The constructor for this class differs from `PathInfo`'s constructor in that it requires one additional argument, the base path.

```
# - - -   B i g I n f o . _ _ i n i t _ _   - - -
def __init__ ( self, path, basePath ):
    """Constructor for BigInfo."""
```

First we call the parent class constructor. Then we store the `basePath` argument in the internal attribute `__basePath`.

```
#-- 1 --
pathinfo.PathInfo.__init__ ( self, path )

#-- 2 --
self.__basePath = basePath
```

7.6. `BigInfo.__cmp__()`: The comparator method

When two instances of the `PathInfo` base class are compared, the `.__cmp__()` method in that class orders them by pathname.

In order to get `BigInfo` objects to sort in descending order by size (with the pathname as a tie-breaker), we redefine that method in this derived class.

```
# - - -   B i g I n f o . _ _ c m p _ _   - - -
def __cmp__ ( self, other ):
    """Compare two BigInfo objects.

    [ other is a BigInfo object ->
      if self should precede other ->
        return a negative number
      else if self should follow other ->
        return a positive number
      else -> return 0 ]
    """
```

To make larger files precede smaller ones, we want to return a negative number if `self.size` is greater than `other.size`, a positive number if it is less, and zero if their `.size` attributes are equal. The `cmp()` function does this comparison, but backwards. So we can implement the comparison we want by inverting the sign of the result of `cmp()`.

We need to consider at more than the file sizes, however. If there are multiple files with the same size, in what order should they be shown? We'll use the pathname as a secondary key. That way, if for example there are a lot of files with length 0, those files will be grouped together but sorted by pathname.

So the first step is to call the `cmp()` function to compare the sizes, and negate its result so we get descending instead of ascending order. If this result is nonzero, we can return it to the caller.

```
#-- 1 --
compare = - cmp ( self.size, other.size )

#-- 2 --
```

```

if compare != 0:
    return compare

```

If the sizes are equal, we then call `cmp()` again on the `.path` attributes, and return that.

bigfiles.py

```

#-- 3 --
return cmp(self.path, other.path)

```

7.7. `BigInfo.__str__()`: String conversion method

If you convert a `PathInfo` object to a string, it starts with the permissions. However, in the `bigfiles.py` script, we're assuming that the user is not going to be interested in permissions, but mainly in the file's size and pathname, and perhaps also its last modification time.

So, to change this format, we can define a `__str__()` method to override the base class's `__str__()` method. This version of the method presents only the modification time, file size, and path name.

There is one refinement to make the display more readable. Because this version of `__str__()` does not include the type code (d for directory, - for regular files), it is hard to tell which pathnames relate to directories. So we append a "/" to the pathname if it is a directory. This is the convention used by the output of the "ls -F" command to identify directories.

bigfiles.py

```

# - - -   B i g I n f o . _ _ s t r _ _   - - -

def __str__( self ):
    """Format a BigInfo for printing."""

```

So that the reader of the report can tell which lines are for directories, we set `suffix` to a slash if this path is a directory, or to an empty string otherwise.

bigfiles.py

```

#-- 1 --
# [ if self represents a directory ->
#     suffix := "/"
# else ->
#     suffix := "" ]
if self.isDir(): suffix = "/"
else:            suffix = ""

```

Next we find the path relative to `self.__basePath`. This code assumes that `self.__basePath` is the absolute path name of a directory above our path. To get the relative path, we can then just use `os.path.abspath()` to get our path's absolute path, then trim off the first `len(self.__basePath)` characters, plus one for the slash that separates those two parts.

bigfiles.py

```

#-- 2 --
# [ self.__basePath is the absolute path of a directory
#   above self.path ->
#     relPath := path to self.path relative to
#               self.__basePath ]
absPath = os.path.abspath ( self.path )
relPath = absPath [ len(self.__basePath) + 1 : ]

```

There is one special case: the first line of the report is for the base path itself, whose absolute path is identical to `self.__basePath`, and `relPath` is now an empty string. In this case, we substitute "." for the path name, and set `suffix` to the empty string so that the line will not read "./".

bigfiles.py

```
#-- 3 --
if relPath == "":
    relPath = "."
    suffix = ""
```

Finally we are ready to format and return the report line.

bigfiles.py

```
#-- 4 --
return ( "%s %10s %s%s" %
        (self.modTime(), self.size, relPath, suffix) )
```

7.8. class `BigReport`: The class for the whole application

The `BigReport` class is a container for all the information we need to produce the report. Its constructor takes the name of a directory, walks that directory subtree, and records all the file information. Its `.genFiles()` method is used to extract the resulting report in the desired order. Here is its interface:

bigfiles.py

```
# - - - - - c l a s s   B i g R e p o r t   - - - - -

class BigReport:
    """Holds the big-files report.

    Exports:
    BigReport ( dir ):
        [ dir is a string ->
          if dir names a directory to which we have access ->
            return a BigReport object describing all the
            accessible files in that directory's subtree
          else -> raise OSError ]
    .genFiles():
        [ generate a sequence of BigInfo objects representing
          the files in self, in descending order by file size,
          with the path name as a secondary key ]

    Class invariants:
    .__bigList:
        [ a list of information on all the files in self
          as BigInfo objects, sorted ]
    """
```

7.9. `BigReport.__init__()`: Constructor

The constructor takes as an argument the name of a directory, and the name of a directory above it so it can display the path name relative to that directory. First we initialize the internal `.__bigList` attribute.

```
# - - -   B i g R e p o r t . _ _ i n i t _ _   - - -

def __init__ ( self, dir ):
    """Constructor for the BigReport class."""

    #-- 1 --
    self.__bigList = []
```

To visit every file in the subtree, we use the `os.path.walk()` function. This function takes three arguments:

1. The name of the directory subtree to be walked.
2. A “visitor function” that will be called once for every directory in the subtree, including the starting directory.
3. The third argument gets passed on to the visitor function as its first argument. We'll use this argument to pass the starting directory name to the visitor function, because the `BigInfo` object needs it to determine relative path names.

```
#-- 2 --
# [ dir is a string ->
#     self.__bigList := self.__bigList with BigInfo
#                     objects added representing every accessible
#                     file in the subtree named by dir ]
os.path.walk ( dir, self.__visitor, dir )
```

All that remains in the constructor is to sort the list.

```
#-- 3 --
# [ self.__bigList := self.__bigList, sorted ]
self.__bigList.sort()
```

7.10. `BigReport.__visitor()`: Visitor function for `os.path.walk()`

When we call `os.path.walk()`, we pass it this method as the “visitor function”. This function is called once for each directory in the subtree. As discussed in the *Python Library Reference*¹⁵ section on the `os.path` module, the visitor function takes three arguments:

1. `arg`: The value passed as the third argument to `os.path.walk()` is passed on to the visitor function. We are not using this value.
2. `dirName`: The name of the directory we are currently visiting.
3. `nameList`: A list of the names within this directory. This may include regular files, subdirectories, and soft links (and perhaps other creatures that are not of interest to this script). If the directory is empty, this argument will be an empty list.

This method must find all the regular files in `nameList`, take snapshots of them with the `BigInfo` constructor, and add those `BigInfo` instances to the `self.__bigList` list.

¹⁵ <http://docs.python.org/lib/module-os.path.html>

We can ignore subdirectories here, because `os.path.walk()` will take care of calling the visitor function for them.

bigfiles.py

```
# - - - B i g R e p o r t . _ _ v i s i t o r - - -

def __visitor ( self, basePath, dirName, nameList ):
    """Visitor function for os.path.walk.

    [ (basePath is the absolute path name to a directory
      above dirName) and
      (dirName is the name of a directory) and
      (nameList is a list of the names within that
      directory) ->
      self.__bigList := self.__bigList with BigInfo
                      objects added representing the accessible
                      ordinary files in nameList ]
    """
```

The first step is to add an entry for the directory itself. It is unlikely that the directory will be inaccessible, but we use a `try:/except:` block just in case it is, so the script won't crash.

bigfiles.py

```
#-- 1 --
# [ self.__bigList := self.__bigList with a BigInfo
#     object added representing dirName ]
try:
    dirInfo = BigInfo ( dirName, basePath )
    self.__bigList.append ( dirInfo )
except OSError, detail:
    pass
```

Next, we iterate through the files in `nameList`, attempting to pass each one to `BigInfo`. If we don't have access to the file, that constructor will raise an `OSError` exception; in that case, we just discard that name and move on to the next one.

Each file's path name must be reconstructed by prepending it with `dirName`. We use the special `os.path.join()` function to concatenate them.

Then, we append each `BigInfo` object to `self.__bigList` only if it is a regular file.

bigfiles.py

```
#-- 2 --
for fileName in nameList:
    #-- 2 body --
    # [ if fileName names an accessible regular file ->
    #     self.__bigList := self.__bigList with a new
    #         BigInfo object representing fileName
    #     else -> I ]

    #-- 2.1 --
    # [ filePath := dirName + fileName ]
    filePath = os.path.join ( dirName, fileName )

    #-- 2.2 --
    # [ if filePath is an accessible path to a regular file ->
    #     self.__bigList := self.__bigList + (a BigInfo
```

```

#         showing the status of filePath)
# else -> I ]
try:
    bigInfo = BigInfo ( filePath, basePath )
    if bigInfo.isFile():
        self.__bigList.append ( bigInfo )
except OSError, detail:
    pass

```

Note the `pass` statement above. This causes inodes such as block and character device files to be ignored silently.

7.11. `BigReport.genFiles()`: Generate the report

This method generates the elements of `.self.__bigList()` in order. To print the report, the caller can just use a `print` statement on each returned value: that will convert it to a string and print it. Then we raise the special `StopIteration` exception to signify the end of generated values.

bigfiles.py

```

# - - -   B i g R e p o r t . g e n F i l e s   - - -

def genFiles ( self ):
    """Generate the BigInfo objects in self.__bigList."""
    for bigInfo in self.__bigList:
        yield bigInfo

    raise StopIteration

```

7.12. Epilogue

These last few lines of the script invoked the `main()`, but only if the script is being run (as opposed to being imported).

bigfiles.py

```

# - - - - -

if __name__ == "__main__":
    main()

```

8. Source code for `oldfiles.py`

This script is generally quite similar to `bigfiles.py`. The only important difference is how the files are sorted: in this case, they are sorted in descending order by modification time, that is, in reverse chronological order.

As with the `bigfiles.py` script, we create a class that inherits from `PathInfo`, and define a different `__cmp__()` method to change the sorting behavior. Like `bigfiles.py`, we also redefine the `__str__()` method to change the format used to display each file's information.

We'll call the derived class `OldInfo`.

So, here's the overall flow for processing each directory tree named on the command line.

1. Instantiate a class called `OldReport` that holds all the information needed to generate our report. The class constructor takes the name of the director as an argument. It walks the directory tree, makes each thing it finds into an `OldInfo` object, and then sorts them.
2. Use the `.genFiles()` method of the `BigReport` object to generate the `OldInfo` objects in reverse chronological order, printing each one as it is generated.

8.1. `oldfiles.py`: Code prologue

This part is identical to the corresponding section of `bigfiles.py`.

`oldfiles.py`

```
#!/usr/bin/env python
#=====
# oldfiles.py: Show files in reverse chron. order by mod. time.
# For documentation in "literate programming" style, see:
#   http://www.nmt.edu/help/lang/python/examples/pathinfo/
#-----

SCRIPT_NAME      = "oldfiles.py"
EXTERNAL_VERSION = "1.1"

#=====
# Imports
#-----
import sys, os
import pathinfo
```

8.2. `oldfiles.py`: The main program

Again, this section is pretty much identical to the main program of `bigfiles.py`.

`oldfiles.py`

```
# - - - - - m a i n - - - - -

def main():
    """Main program."""

    print "=== %s %s ===" % (SCRIPT_NAME, EXTERNAL_VERSION)

    #-- 1 --
    # [ if sys.argv[1:] is empty ->
    #   dirList := [ "." ]
    # else ->
    #   dirList := sys.argv[1:] ]
    dirList = sys.argv[1:]
    if len(dirList) == 0:
        dirList = [ "." ]

    #-- 2 --
    # [ sys.stdout += reports listing files below each
    #   directory named in dirList, with files in reverse
    #   chronological order by modification time ]
```

```

for dirName in dirList:
    #-- 2 body --
    # [ dirName is a string ->
    #     sys.stdout += a report listing the files below
    #         directory (dirName), with files in reverse
    #         chronological order by modification time ]
    report ( dirName )

```

8.3. report (): Generate one directory tree's report

This function generates the report for one directory subtree. The pathname of the directory is the argument.

oldfiles.py

```

# - - -   r e p o r t   - - -

def report ( dirName ):
    """Generate the report for one directory subtree.

    [ dirName is a string ->
      sys.stdout += a report listing the files below
      directory (dirName), with files in reverse
      chronological order by modification time ]
    """

    #-- 1 -
    # [ basePath := dirName's absolute path name
    #     sys.stdout += report heading showing dirName's real
    #         absolute path ]
    basePath = os.path.abspath ( dirName )
    print "\n   === %s ===" % os.path.realpath ( dirName )

    #-- 2 --
    # [ oldReport := an OldReport object describing all the
    #     accessible files in directory tree (dirName) ]
    oldReport = OldReport ( basePath )

    #-- 3 --
    # [ oldReport is an OldReport object ->
    #     sys.stdout += lines describing files in oldReport
    #         in chronological order by modification time ]
    for oldInfo in oldReport.genFiles():
        print oldInfo

```

8.4. class OldInfo: The PathInfo subclass

This class is very similar to the BigInfo class in bigfiles.py see Section 7.4, “class BigInfo: The PathInfo subclass” (p. 19). It too inherits from the PathInfo class.

oldfiles.py

```

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

```

```
# - - - - - c l a s s   O l d I n f o   - - - - -

class OldInfo(pathinfo.PathInfo):
    """Represents information about one file; sorts by mod time.
    """
```

8.5. OldInfo.__init__()

This constructor is similar to Section 7.5, “BigInfo.__init__(): Constructor” (p. 19).

oldfiles.py

```
# - - -   O l d I n f o .   _ _   i n i t   _ _   - - -

def __init__ ( self, path, basePath ):
    """Constructor for OldInfo."""

    #-- 1 --
    pathinfo.PathInfo.__init__ ( self, path )

    #-- 2 --
    self.__basePath = basePath
```

8.6. OldInfo.__cmp__(): The comparator method

This method defines how comparison operators work when comparing two OldInfo objects. The returned value uses the same conventions as Python’s built-in cmp() function: it returns a negative number if the first argument should come first, a positive number if the second argument should come first, and zero if they are considered equal.

To get this effect, we can simply call the cmp() function and pass it the two timestamps. The result of that function would order the objects in chronological order, so we could negate that result to get reverse chronological order. Or, as is done here, we could just reverse the order of the arguments.

oldfiles.py

```
# - - -   O l d I n f o .   _ _   c m p   _ _   - - -

def __cmp__ ( self, other ):
    """Compare two OldInfo objects.

    [ other is an OldInfo object ->
      if self should precede other ->
        return a negative number
      else if self should follow other ->
        return a positive number
      else -> return 0 ]
    """
    return cmp ( other.modEpoch, self.modEpoch )
```

8.7. OldInfo.__str__(): String conversion method

This method is completely identical to the corresponding method in the `bigfiles.py` script; see Section 7.7, “BigInfo.__str__(): String conversion method” (p. 21).

oldfiles.py

```
# - - -   O l d I n f o .   _ _ s t r _ _   - - -

def __str__( self ):
    """Format an OldInfo object for printing."""

    #-- 1 --
    # [ if self represents a directory ->
    #     suffix := "/"
    # else ->
    #     suffix := "" ]
    if self.isDir(): suffix = "/"
    else:            suffix = ""

    #-- 2 --
    # [ self.__basePath is the absolute path of a directory
    #   above self.path ->
    #     relPath := path to self.path relative to
    #               self.__basePath ]
    absPath = os.path.abspath ( self.path )
    relPath = absPath [ len(self.__basePath) + 1 : ]

    #-- 3 --
    if relPath == "":
        relPath = "."
        suffix = ""

    #-- 4 --
    return ( "%s %10s %s%s" %
            (self.modTime(), self.size, relPath, suffix) )
```

8.8. class OldReport: The application class

This class is a container for a list of OldInfo objects. Because of the behavior of the OldInfo.__cmp__(), sorting this list puts the entries into reverse chronological order.

The constructor takes the pathname of the directory subtree as an argument, builds the list of OldInfo objects, and sorts them. You can call the .genFiles() method and it will generate the contained OldInfo objects in reverse chronological order.

Again, this class is just about identical to the corresponding class in the `bigfiles.py` script; see Section 7.8, “class BigReport: The class for the whole application” (p. 22).

oldfiles.py

```
# - - - - -   c l a s s   O l d R e p o r t   - - - - -

class OldReport:
    """Holds the old-files report.

    Exports:
```

```

OldReport ( dir ):
    [ dir is a string ->
      if dir names a directory to which we have access ->
        return an OldReport object describing all the
        accessible files in that directory's subtree
      else -> raise OSError ]
.genFiles():
    [ generate a sequence of OldInfo objects representing
      the files in self, in reverse chronological order
      by modification timestamp ]

Class invariants:
    .__oldList:
        [ a list of information on all the files in self
          as OldInfo objects, sorted ]
    """

```

8.9. OldReport.__init__(): Constructor

The sole argument is the name of the directory. The constructor creates an empty list `.__oldList`, fills it with `OldInfo` objects made from the directories and files in the given subtree, and then sorts them.

For a more detailed description of the steps here, refer to the nearly identical method, Section 7.9, “`BigReport.__init__(): Constructor`” (p. 22).

oldfiles.py

```

# - - - O l d R e p o r t . _ _ i n i t _ _ - - -

def __init__ ( self, dir ):
    """Constructor for the OldReport class."""

    #-- 1 --
    self.__oldList = []

    #-- 2 --
    # [ dir is a string ->
    #   self.__oldList := self.__oldList with OldInfo
    #   objects added representing every accessible
    #   file in the subtree named by dir ]
    os.path.walk ( dir, self.__visitor, dir )

    #-- 3 --
    # [ self.__oldList := self.__oldList, sorted ]
    self.__oldList.sort()

```

8.10. OldReport.__visitor(): Visitor function for os.path.walk()

Very similar to Section 7.10, “`BigReport.__visitor(): Visitor function for os.path.walk()`” (p. 23).

oldfiles.py

```

# - - - O l d R e p o r t . _ _ v i s i t o r - - -

```

```

def __visitor ( self, basePath, dirName, nameList ):
    """Visitor function for os.path.walk.

    [ (dirName is the name of a directory) and
      (nameList is a list of the names within that
      directory) ->
      self.__oldList := self.__oldList with BigInfo
      objects added representing the accessible
      ordinary files in nameList ]
    """

    #-- 1 --
    # [ self.__oldList := self.__oldList with an OldInfo
    #   object added representing dirName ]
    try:
        dirInfo = OldInfo ( dirName, basePath )
        self.__oldList.append ( dirInfo )
    except OSError, detail:
        pass

    #-- 2 --
    for fileName in nameList:
        #-- 2 body --
        # [ if fileName names an accessible regular file ->
        #   self.__oldList := self.__oldList with a new
        #   OldInfo object representing fileName
        #   else -> I ]

        #-- 2.1 --
        # [ filePath := dirName + fileName ]
        filePath = os.path.join ( dirName, fileName )

        #-- 2.2 --
        # [ if filePath is an accessible path to a regular file ->
        #   self.__oldList := self.__oldList + (an OldInfo
        #   showing the status of filePath)
        #   else -> I ]
        try:
            oldInfo = OldInfo ( filePath, basePath )
            if oldInfo.isFile():
                self.__oldList.append ( oldInfo )
        except OSError, detail:
            pass

```

8.11. OldReport.genFiles(): Generate the report

For a detailed narrative, see the identical method in `bigfiles.py`: Section 8.11, “OldReport.genFiles(): Generate the report” (p. 31).

`oldfiles.py`

```

# - - - O l d R e p o r t . g e n F i l e s - - -

def genFiles ( self ):

```

```

        """Generate the OldInfo objects in self.__oldList."""
        for oldInfo in self.__oldList:
            yield oldInfo

        raise StopIteration

```

8.12. Epilogue

These lines call `main()`, assuming that this is run as a script.

oldfiles.py

```

# -----

if __name__ == "__main__":
    main()

```

9. Source code for `softlinks.py`

The overall flow of this script is similar to that for `bigfiles.py`. If any directory paths are given on the command line, a report is printed for each one. If there are no command line arguments, one report is printed for the current directory `."`.

As the `os.path.walk()` function visits every directory and file in a tree, the visitor function finds the soft links, and accumulates a list of `PathInfo` instances for each one. Those instances are sorted by path name (which is the default ordering for `PathInfo` instances), then the entries in the sorted list are displayed in the body of the report.

9.1. `softlinks.py`: Code prologue

The script starts with the usual "pound bang line", a comment pointing to this documentation, and constants defining the script name and external version number.

softlinks.py

```

#!/usr/bin/env python
#=====
# softlinks.py:  Script to find all soft links a directory tree.
#   For documentation in "literate programming" style, see:
#       http://www.nmt.edu/tcc/help/lang/python/examples/pathinfo/
#-----

SCRIPT_NAME = "softlinks.py"
EXTERNAL_VERSION = "1.0"

```

Next we import the usual modules for the system and operating system interfaces, and of course the `PathInfo` class.

softlinks.py

```

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

import sys, os
import pathinfo

```

9.2. main(): Main program

The main starts by building the list of directories from the command line, defaulting to ["."] if there are no command line arguments.

softlinks.py

```
# - - - - - m a i n - - - - -

def main():
    """Main program."""
    print "=== %s %s ===" % (SCRIPT_NAME, EXTERNAL_VERSION)

    #-- 1 --
    # [ if sys.argv[1:] is empty ->
    #     dirList := [ "." ]
    # else ->
    #     dirList := sys.argv[1:] ]
    dirList = sys.argv[1:]
    if len(dirList) == 0:
        dirList = [ "." ]
```

Next we generate a report for each member of dirList.

softlinks.py

```
#-- 2 --
# [ sys.stdout += reports listing links below each
#     directory named in dirList, with files in
#     ascending order by pathname ]
for dirName in dirList:
    #-- 2 body --
    # [ dirName is a string ->
    #     sys.stdout += a report listing the files below
    #     directory (dirName), with files in ascending
    #     order by pathname ]
    report ( dirName )
```

9.3. report(): Generate one tree's report

This function generates the portion of the output for one directory subtree.

softlinks.py

```
# - - - r e p o r t - - -

def report ( dirName ):
    """Generate the report for one directory subtree.

    [ dirName is a string ->
      sys.stdout += a report listing the files below
      directory (dirName), with files in ascending
      order by pathname ]
    """
```

At this level, the logic is broken into three steps: write the report header; instantiate a `LinkReport` object containing information on all the links in the subtree; and call the `LinkReport.genLinks()` method to generate the lines of the report.

```

#-- 1 --
# [ basePath := dirName's absolute path name
#   sys.stdout += report heading showing dirName's real
#               absolute path ]
basePath = os.path.abspath ( dirName )
print "\n   == %s ==" % os.path.realpath ( dirName )

```

The `basePath` is the path to the root directory of the subtree, so it will be a prefix to every pathname under it. The report will remove this prefix, showing each link's path name relative to `basePath`. For details of the report format, see Section 5, “`softlinks.py`: Find soft links in a directory tree” (p. 7).

```

#-- 2 --
# [ linkReport := a LinkReport instance describing all the
#               accessible soft links in directory tree (basePath) ]
linkReport = LinkReport ( basePath )

```

A `LinkReport` instance is basically a container for `LinkInfo` instances. The `LinkReport.genLinks()` method generates these `LinkInfo` instances in sorted order. Formatting the report is handled by the `__str__()` special method in the `LinkInfo` class.

```

#-- 3 --
# [ linkReport is a LinkReport instance ->
#   sys.stdout += lines describing links in linkReport
#               in ascending order by path name ]
for linkInfo in linkReport.genLinks():
    print linkInfo

```

9.4. class LinkReport: Link report container

An instance of `LinkReport` is a container for information about the links in a given directory subtree. The information about each link is represented as a `LinkInfo` instance; see Section 9.8, “class `LinkInfo`: The `PathInfo` subclass” (p. 36).

```

# - - - - -   c l a s s   L i n k R e p o r t   - - - - -

class LinkReport:
    """Holds the links report for one directory subtree.

    Exports:
    LinkReport ( dir ):
        [ dir is a string ->
          if dir names a directory to which we have access ->
            return a LinkReport object describing all the
            accessible links in that directory's subtree
          else -> raise OSError ]
    .genLinks():
        [ generate the links in self as a sequence of
          LinkInfo instances, in ascending order by path name ]

    Class invariants:

```

```

    ..__linkList:
        [ a list of LinkInfo objects representing the soft
          links in self ]
    """

```

9.5. LinkReport.__init__(): Constructor

The constructor starts by initializing `self.__linkList` as an empty list.

softlinks.py

```

# - - -   L i n k R e p o r t .   _ _ i n i t   _ _   - - -

def __init__ ( self, dir ):
    """Constructor for the LinkReport class."""

    #-- 1 --
    self.__linkList = []

```

The `os.path.walk` function calls our `.__visitor()` method once for each directory in and under `dir`, passing it a list of the names in that directory; see Section 9.6, “`LinkReport.__visitor():` Visitor function for `os.path.walk()`” (p. 35). The end result is to add one `LinkInfo` instance to `self.__linkList` for each soft link in the tree.

softlinks.py

```

#-- 2 --
# [ dir is a string ->
#     self.__linkList := self.__linkList with LinkInfo
#     instances added representing every accessible
#     soft link in the subtree rooted in dir ]
os.path.walk ( dir, self.__visitor, dir )

```

All that remains is to sort the links by path name.

softlinks.py

```

#-- 3 --
# [ self.__linkList := self.__linkList, sorted ]
self.__linkList.sort()

```

9.6. LinkReport.__visitor(): Visitor function for os.path.walk()

This function is called once for each directory in the subtree. The first argument is the base pathname for the subtree, so that we can generate path names relative to there. The second argument is the directory we are visiting. The third argument is a list of all the names in the directory.

softlinks.py

```

# - - -   L i n k R e p o r t .   _ _ v i s i t o r   - - -

def __visitor ( self, basePath, dirName, nameList ):
    """Visitor function for os.path.walk.

    [ (basePath is the absolute path to a directory
      at or above dirName) and
      (dirName is the name of a directory) and
      (nameList is a list of the names in that directory) ->
      self.__linkList := self.__linkList with LinkInfo
    """

```

```

        instances added representing accessible links
        in nameList ]
    """

```

The names in `nameList` are the files and soft links in this directory. We iterate through that list, adding more `LinkInfo` instances to `self.__linkList` corresponding to accessible soft links. The full path to each file is reconstructed by using the `os.path.join()` function to prepend `dirName`.

softlinks.py

```

#-- 1 --
for fileName in nameList:
    #-- 1 body --
    # [ if fileName names an accessible soft link ->
    #     self.__linkList := self.__linkList + (a new
    #         LinkInfo instance describing fileName)
    #     else -> I ]

    #-- 1.1 --
    # [ filePath := dirName + fileName ]
    filePath = os.path.join ( dirName, fileName )

    #-- 1.2 --
    # [ if filePath is an accessible soft link ->
    #     self.__linkList := self.__linkList + (a new
    #         LinkInfo instance describing filePath) ]
    try:
        linkInfo = LinkInfo ( filePath, basePath )
        if linkInfo.isLink():
            self.__linkList.append ( linkInfo )
    except OSError, detail:
        pass

```

9.7. `LinkReport.genLinks()`: Generate links

softlinks.py

```

# - - - L i n k R e p o r t . g e n L i n k s - - -

def genLinks ( self ):
    """Generate the LinkInfo instances in self."""
    for linkInfo in self.__linkList:
        yield linkInfo

    raise StopIteration

```

9.8. `class LinkInfo`: The `PathInfo` subclass

This class is derived from `PathInfo`. It takes an additional constructor argument: the base path name for the directory subtree, necessary so we can display the path name for this link relative to that base path name.

softlinks.py

```

# - - - - - c l a s s L i n k I n f o - - - - -

```

```

class LinkInfo(pathinfo.PathInfo):
    """Represents information about one file; sorts by path name.

    Exports (other than those inherited):
    LinkInfo ( path, basePath ):
        [ (path is the path name to a file) and
          (basePath is the path name of a directory above path) ->
          return a new LinkInfo instance with those values ]
    .__str__ ( self ):
        [ return a multiline string describing self's path,
          whether or not it is a broken link, and where it
          points ]

    State/Invariants:
    .__basePath:      [ as passed to constructor ]
    """

```

9.9. LinkInfo.__init__(): Constructor

This constructor first calls the parent class's constructor, then stores away the `basePath` argument within the instance.

softlinks.py

```

# - - -   L i n k I n f o . _ _ i n i t _ _   - - -

def __init__ ( self, path, basePath ):
    """Constructor for LinkInfo."""

    #-- 1 --
    pathinfo.PathInfo.__init__ ( self, path )

    #-- 2 --
    self.__basePath = basePath

```

9.10. LinkInfo.__str__(): Convert to a string

This method returns a string containing two lines of output. The format is described in Section 5, "softlinks.py: Find soft links in a directory tree" (p. 7). The class variables `goodPrefix` and `badPrefix` should be the same length; the former is used for links to existing files, the latter for broken links.

softlinks.py

```

# - - -   L i n k I n f o . _ _ s t r _ _   - - -

goodPrefix    = "      "
brokenPrefix  = "   #### "

def __str__ ( self ):
    """Convert a LinkInfo instance to a string.
    """

```

First we convert the link's absolute path name to a relative path name by removing `self.__basePath` from the front of it. This relies on the precondition that `self.__basePath` is a directory above `self.path`.

softlinks.py

```
#-- 1 --
# [ fullPath := self's absolute path without links
#           replaced
#   targetPath := self's target path ]
fullPath = self.absPath()
targetPath = self.realPath()

#-- 2 --
# [ self.__basePath is a directory above fullPath ->
#   relPath := path to fullPath relative to
#           self.__basePath ]
relPath = fullPath [ len(self.__basePath) + 1 : ]
```

Next we use `os.path.exists()` to find out whether the link is broken or not, and set `prefix` to indicate whether it is broken. Finally we format the two-line report string and return it.

softlinks.py

```
#-- 2 --
# [ if targetPath exists ->
#   prefix := self.goodPrefix
#   else ->
#   prefix := self.brokenPrefix ]
if os.path.exists ( targetPath ):
    prefix = self.goodPrefix
else:
    prefix = self.brokenPrefix

#-- 3 --
return ( "%s ->\n%s%s" %
        (relPath, prefix, targetPath) )
```

9.11. Epilogue

softlinks.py

```
# - - - - -

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