

webstats.py 4.0: Internal Maintenance Specification



John W. Shipman

2011-11-11 17:25

Abstract

Describes a system that generates reports summarizing web page access counts on the New Mexico Tech Computer Center web server, <http://www.nmt.edu/>.

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	3
2. Overview	3
2.1. Data structures	3
2.2. Overall program flow	4
2.3. Navigational considerations	5
3. The webstats.py module: Main program	5
4. The webstats.py file: Prologue	6
5. Imported modules	6
6. Manifest constants	7
6.1. EXPIRE_DAYS	7
6.2. Input file paths	7
6.3. Web paths	8
6.4. Constants for HTML generation	10
6.5. Report label text	10
7. Main program	11
8. inputPhase(): Read the access logs	13
9. readLogFile(): Process one access log	15
10. buildAllPages(): Generate all output pages	15
11. addSummaryTable(): Generate the table summarizing all accesses	17
12. buildHitParade(): Build the hit parade	18
13. accessReport(): Start a new access report table	19
14. accessRow(): Add one row to an access report table	20
15. instituteHomepage(): Access report for "/"	21
16. buildCategoryTable(): Build categories table and all personal and official reports	21
17. buildPersonalSide(): Letter and personal pages	23
18. buildLetter(): Build one letter page and related personal pages	24
19. addPersonalReport(): Generate links and access report for one personal account	25
20. buildReportPage(): Build one access report page	26
21. buildOfficialSide(): Build access reports for official directories	28

¹ <http://www.nmt.edu/tcc/projects/tccwebstats4/ims/>

² <http://www.nmt.edu/tcc/projects/tccwebstats4/ims/webstatsims.pdf>

22. <code>fatal()</code> : Write a message and stop	29
23. <code>class AccessSummary</code> : Principal data structure	29
24. Manifest constants for <code>class AccessSummary</code>	31
24.1. <code>AccessSummary.EXPIRE_DAYS</code> : Duration of the report interval	32
24.2. <code>AccessSummary.SYM_DOMAIN</code> : Local domain name, symbolic form	32
24.3. <code>AccessSummary.IP_DOMAIN</code> : Local domain in dotted form	32
24.4. <code>AccessSummary.BAD_STATUS_THRESHOLD</code> : Upper limit for status codes	32
24.5. <code>AccessSummary.IGNORED_EXTENSIONS</code> : File extensions to be ignored	32
24.6. <code>AccessSummary.SPIDER_STRINGS</code> : Spider detection strings	32
25. <code>AccessSummary.__init__()</code> : Constructor	33
26. <code>AccessSummary.addPageGet()</code> : Process one access record	33
27. <code>AccessSummary.__isRelevant()</code> : Filter out irrelevant access records	34
28. <code>AccessSummary.__statusFilter()</code> : Filter by status code	34
29. <code>AccessSummary.__extFilter()</code> : Ignore certain files by extension	35
30. <code>AccessSummary.__spiderFilter()</code> : Filter out search engine spider accesses	35
31. <code>AccessSummary.__pwdFilter()</code> : Filter out password-protected pages	36
32. <code>AccessSummary.__timeFilter()</code> : Filter out expired records	36
33. <code>AccessSummary.__specialFilter()</code> : Special case filter	37
34. <code>AccessSummary.FILTER_FUNCTIONS</code> : Collection of filter functions	38
35. <code>AccessSummary.__addHit()</code> : Register one access	38
36. <code>AccessSummary.__addUrl()</code> : Register one access in <code>self.__urlMap</code>	39
37. <code>AccessSummary.__addCategory()</code> : Add URL to appropriate category	40
38. <code>AccessSummary.getUrl()</code> : Retrieve hit counts for a given URL	42
39. <code>AccessSummary.genByHits()</code> : Generate the hit parade	42
40. <code>AccessSummary.genPersonalLetters()</code> : First letters of personal accounts	43
41. <code>AccessSummary.genPersonals()</code> : Generate accounts with the same first letter	43
42. <code>AccessSummary.genOfficials()</code> : Generate names of official directories	44
43. <code>AccessSummary.genPersonUrls()</code> : All URLs for a given person	44
44. <code>AccessSummary.genOfficialUrls()</code> : All URLs for an official directory	45
45. <code>class HitCount</code> : Hit counts for one URL	45
46. <code>HitCount.__init__()</code> : Constructor	46
47. <code>HitCount.addHit()</code> : Tally one access	46
48. <code>Hitcount.__cmp__()</code> : Comparator method	46
49. <code>Hitcount.__lt__()</code> : Less-than method	47
50. Epilogue	47
51. The <code>pageget.py</code> module: Apache log file functions	47
51.1. Prologue to <code>pageget.py</code>	47
51.2. <code>class FixedTimeZone</code>	48
51.3. <code>scanAccessLog()</code> : Scan an access log file	49
51.4. <code>scanAccessLine()</code> : Process one access log line	50
51.5. Breaking down the log line: a mixed-strategy parse	53
51.6. <code>scanGroups()</code> : Top-level disassembly of the log line	53
51.7. <code>scanQuoted()</code> : Process double-quoted string with escapes	55
51.8. <code>scanAccessGroup()</code> : Process accessors	56
51.9. <code>findHostList()</code> : Derive the effective host list	58
51.10. <code>scanDateGroup()</code> : Process date	59
51.11. <code>scanCmdGroup</code> : Process command group	62
51.12. <code>cleanURL()</code> : Process the raw URL	63
51.13. <code>asciifyString</code> : Encode non-ASCII characters	64
51.14. <code>asciifyChar()</code> : Escape a non-ASCII character	65
51.15. <code>scanTailGroup()</code> : Process remaining fields	65
51.16. <code>class PageGet</code> : Describes one page access	66

51.17. <code>PageGet.__init__()</code> : Constructor	67
51.18. <code>PageGet.isFar()</code> : Is this an off-campus accessor?	68
51.19. <code>PageGet.__str__()</code> : Debug display	69
51.20. <code>error()</code> : Write a message to <code>stderr</code>	69
51.21. <code>message()</code> : Send a message to standard error and log file	70
51.22. A small test driver for <code>PageGet</code>	70

1. Introduction

This document describes the `webstats.py` script for reducing and displaying statistics on page accesses from the Tech Computer Center's web server. See the specification³ for the externals of this script.

The code is documented in the “literate programming” style: the source file contains both the documentation and the script's source code. For more information on literate programming and the tool used to extract the source code, see *A source extractor for lightweight literate programming*⁴.

This publication is available in Web form⁵ and also as a PDF document⁶.

2. Overview

The script is written in the Python language. For more information about Python, see *Python 2.2 quick reference*⁷.

These source files comprise the `webstats.py` application:

- `webstats.py`⁸ is the main script.
- The `pageget.py`⁹ module encapsulates the logic for scanning Apache's access log files. See Section 51, “The `pageget.py` module: Apache log file functions” (p. 47).

2.1. Data structures

The `pageget.py` module defines a class `PageGet`, each instance of which represents one line in the Apache server's `access_log` file.

What we need to know from the access logs boils down to three data for each page accessed in the time period of interest. We'll encapsulate these three items as instances of class `HitCount`:

- `.nTotal`: The total number of hits on the page.
- `.nFar`: The number of hits from off-campus, so that we can compute the percentage of off-campus hits.
- `.url`: The page's URL, minus the “`http://infohost.nmt.edu`” prefix.

The `HitCount` class will include a `.__cmp__()` method that sorts instances in hit-parade order: descending order by their `.nTotal` attribute, with their `.url` attribute in ascending order as a secondary key.

³ <http://www.nmt.edu/tcc/projects/tccwebstats/>

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

⁵ <http://www.nmt.edu/tcc/projects/tccwebstats/ims/>

⁶ <http://www.nmt.edu/tcc/projects/tccwebstats/ims/webstatsims.pdf>

⁷ <http://www.nmt.edu/tcc/help/pubs/python22/>

⁸ <http://www.nmt.edu/tcc/projects/tccwebstats4/ims/webstats.py>

⁹ <http://www.nmt.edu/tcc/projects/tccwebstats4/ims/pageget.py>

We'll need a container class to hold the `HitCount` instances for each URL accessed in the report period. The structure and function of this container is driven by the needs of the script:

- In order to generate the overall summary block on the root page, we'll need to remember the range of timestamps included in the report, and the total number of on- and off-campus hits.
- For the access report on the NMT homepage, we'll need a `HitCount` instance for URL `"/"`.
- For the hit parade, we'll need to visit the `HitCount` instances for every URL accessed in the report period, sorted as defined by the `HitCount.__cmp__()` method: in descending order by hit count, with URL as a secondary key.
- To generate the tables of personal and official pages by their first character, we'll need to interrogate the first character values that occur in both those categories.
- To generate the pages showing all the personal and official directories for a given first character, we'll need to interrogate the list of all personal and official URLs that start with a given character.
- To generate the access report pages for a single person or official directory, we'll need to interrogate the list of all URLs that start with given first component (either `"/~login"` or `"/directory"`).
- Finally, to produce the detail lines on the individual access report pages, we'll need to be able to retrieve the `HitCount` instance for any specified URL.

We'll encapsulate all these data and methods as a single instance of class `AccessSummary`. This instance will accept a stream of `PageGet` instances, filter out the ones that don't matter (such as image files and CSS stylesheets), and store the access data as a set of `HitCount` instances in such a way that they can be retrieved by all the access methods described above.

An earlier version of this program used external files and an ancient sort-merge algorithm that dates from the 1960s. However, starting with this version, we can probably fit into memory everything we need. Preliminary testing in January 2009 showed that the number of distinct URLs to be managed is on the order of 100,000. Even with an average URL length of 50 characters or so, that's a minimal memory footprint on the order of 5MB, hardly a strain on today's multi-gigabyte processor memories.

Python's `__slots__` feature, part of the features of new-style classes, allows us to reduce the memory requirements for each of the `HitCount` instances. The named slots will be `.nTotal`, `.nFar`, and `.url`.

2.2. Overall program flow

Given the data structures described in Section 2.1, "Data structures" (p. 3), here is the overall program flow.

1. Determine the reporting interval by subtracting `EXPIRE_DAYS` days from the current time.
2. Create the single `AccessSummary` instance and inform it of the reporting interval.
3. Read all the access logs, generating a stream of records as `PageGet` instances. Select the relevant records using various filtering criteria (see the section "Filtering the Logs" in the specification¹⁰), and add the selected records to the `AccessSummary` instance.
4. Generate the index page and the two 26-entry indices for personal and official pages, by their initial characters.
5. Use the sorted values from the `urlMap` dictionary to build the hit parade page, minus any entries with fewer than `MIN_HITS` accesses.
6. Generate the two-column table of links to personal and official pages.

¹⁰ <http://www.nmt.edu/tcc/projects/tccwebstats4/>

For each link in this table, also generate the pages with links to each user (`pers-a.html`, `pers-b.html` and so on, and `off-a.html`, `off-b.html`, and so on.

While generating *those* pages, also generate the access report pages for each user or official directory.

2.3. Navigational considerations

Our general standards for Web page navigational links is given in the *TCC Documentation Guidelines*¹¹.

Since the pages we generate will be linked to from the PyStyler part of the TCC web, they should be similar in style. Hence we will use the module described in *tccpage2.py: Dynamic generation of TCC-style web pages with lxml*¹².

The exact sequence of top and bottom links is flexible, and depends on the expected user workflow, especially for dynamically generated pages such as the ones we are building. Here are the expected use cases for our structure:

- The author of a set of Web pages wants to see which pages are actually being visited. From the index page, they will click on the index of personal (or official) pages, then the first letter of the account or directory name, then on the account or directory name.
- An author wants to see which of their pages are the most popular. They will start on the hit parade page, and use their browser's text search function to search for their URL. This search will visit their most popular pages in descending order of popularity.
- A user has an idle curiosity about what pages are popular on this server. They will click to the hit parade.

Here are the navigational features we'll use, and where they will lead.

- *Next*: Next page in sequence. None of our use cases really call for sequential visits to either the letter index pages or the personal or official pages. This will be a dummy link.
- For the same reason, the *Previous* button will be a dummy link.
- There should be a link that gives the user a place to jump up in the structure. For all subpages of the index page, even the individual personal or official URL reports, this link will go back to the index page and have the text *Web server statistics*. For the index page, this link will be missing, as it is covered by the next link type.
- For those who are really lost, we'll provide a *Tech Computer Center* link that goes to the TCC homepage.

3. The `webstats.py` module: Main program

Module `webstats.py` is the top-level script for this application.

The main script starts with a brief prologue. This is major version 4.0.

- The first version was discarded when the logs got too large for in-memory data structures. It was called `weblog` long before the term “weblog” was current.
- The second version was obsoleted by changes in the configuration of the web server, and the author found the documentation badly out of sync with the code. It was called `webstats`, but this name may be a copyright infringement, based on a brief Google search.

¹¹ <http://www.nmt.edu/tcc/doc/plan/>

¹² <http://infohost.nmt.edu/tcc/projects/tccpage2/>

- The third version was a fairly thorough rewrite. In addition to providing another literate programming example, which should make the program more maintainable, it addressed a few minor problems such as the failure to parse correctly access log entries that have escaped double-quote characters inside double-quoted strings. Also, the web pages were generated in XHTML using the techniques described in *Python and the XML Document Object Model (DOM)*¹³.
- The fourth version was triggered by the upgrade of the `infohost` server in January 2009. On this server, the access logs were rotated weekly, not daily. By this point, processor memories were sufficiently large to allow all the input data to reside in memory, obviating the need for the external sort-merge representation of access data used in the previous major version.

The author chose a fairly thorough rewrite in order to improve the navigation: the previous version had one huge page with links to all personal and official reports, and moving to a thumb-index by first letter makes it quicker for a user to find their report.

Python's XML libraries have also improved a lot since the previous version, and the use of the `lxml` library and the `etbuilder.py` module, based on Fredrik Lundh's work, greatly simplify and clarify the generation of web pages.

4. The `webstats.py` file: Prologue

Here is the beginning of the actual code for `webstats.py`. It starts with the usual Unix “pound-bang line” that makes the script self-executing.

```
webstats.py
```

```
#!/usr/bin/env python
#=====
# webstats.py:  NM Tech Computer Center web statistics
#
# For documentation, see:
#   http://www.nmt.edu/tcc/projects/tccwebstats/      Specification
#   http://www.nmt.edu/tcc/projects/tccwebstats/ims/  Internals
#-----
PRODUCT_NAME      = "tccwebstats"
EXTERNAL_VERSION  = "4.0"
DATE_FORMAT       = "%Y-%m-%dT%H:%M:%S %Z"
```

5. Imported modules

Next are the module imports. We'll need the `sys` module for standard streams and the `os` module for operating system functions. We need `math` for its `ceil()` function.

```
webstats.py
```

```
#=====
# Imports
#-----
import sys
import os
import math
```

¹³ <http://www.nmt.edu/tcc/help/pubs/pyxml>

The `datetime` module has a full range of clock and calendar functions.

webstats.py

```
import datetime
```

The general technique for generating XML is described in *Python XML processing with lxml*¹⁴.

webstats.py

```
from etbuilder import E, et
```

The `tccpage2` module assists in building Web pages that conform to the TCC style.

webstats.py

```
import tccpage2
```

The only module specific to this application is `pageget.py`, which contains all the logic related to processing Apache access logs; see Section 51, “The `pageget.py` module: Apache log file functions” (p. 47).

webstats.py

```
from pageget import *
```

6. Manifest constants

For ease of maintenance, a number of manifest constants are declared at the top of the script.

webstats.py

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

6.1. EXPIRE_DAYS

Defines the length of the report interval in days.

webstats.py

```
EXPIRE_DAYS = 30
```

6.2. Input file paths

These constants point to the locations of input files. `ACCESS_LOGS_DIR` is the absolute path name of the directory where all the access logs reside. Note that this path exists only on `infohost.nmt.edu`.

webstats.py

```
ACCESS_LOGS_DIR = "/var/log/httpd/"
```

Within that directory, there is one current access log, and zero or more older logs, whose names are the same as the current access log plus extensions “.1”, “.2”, and so on.

webstats.py

```
ACCESS_LOG_PATH = ACCESS_LOGS_DIR + "access_log"
```

`DAYS_PER_LOG` defines the number of days in each older log, currently rotated weekly.

¹⁴ <http://www.nmt.edu/tcc/help/pubs/pylxml>

```
DAYS_PER_LOG = 7
```

`N_OLDER_LOGS` defines the minimum number of older logs we must read to be sure that we'll have `EXPIRE_DAYS` of data.

```
N_OLDER_LOGS = int ( math.ceil ( float ( EXPIRE_DAYS ) /
                                DAYS_PER_LOG ) )
```

`HITS_CUTOFF` defines the minimum number of hits a page needs to appear on the hit parade report.

```
HITS_CUTOFF = 10
```

6.3. Web paths

The constants in this section describe the locations of the various files to be generated.

For each generate file, we need to know its address in two different namespaces:

- When we generate files, we use their absolute path names, so that the script does not need to be run in any particular place.
- Every generated page must display its absolute URL in order to conform to the TCC style guide.

To assist us in keeping these relationships straight, here is a small helper class.

```
# - - - - - c l a s s   W e b P a t h

class WebPath:
    '''Represents an absolute path accessible via the Web.

    Exports:
    WebPath ( url, absPath ):
        [ (url is a URL as a string) and
          (absPath is the equivalent absolute path as a string) ->
          return a new WebPath instance with those values ]
    .url:      [ as passed to constructor, read-only ]
    .absPath:  [ as passed to constructor, read-only ]
```

To generate a new `WebPath` instance that is a subdirectory or file under an existing `WebPath` instance, use this method:

```
.relative(relPath):
    [ relPath is a relative path as a string ->
      return a new WebPath instance representing relPath
      relative to self ]
...

```

The class constructor does nothing but save its arguments.

```
# - - -   W e b P a t h . _ _ i n i t _ _

def __init__ ( self, url, absPath ):
    '''Constructor
```

```

    ...
    self.url = url
    self.absPath = absPath

```

Computation of a new path relative to `self` uses the standard Python `os.path` module¹⁵. The `.join()` function handles concatenation of two pieces of a file name.

```

# - - -   W e b P a t h . r e l a t i v e
webstats.py

def relative ( self, relPath ):
    '''Return a new WebPath relative to self.
    ...
    return WebPath ( os.path.join ( self.url, relPath ),
                    os.path.join ( self.absPath, relPath ) )

```

Here are the constant paths for this application. First, the starting point for all files hosted on the `infohost.nmt.edu` server:

```

webstats.py

NMT_WEB_PATH = WebPath ( "http://www.nmt.edu/",
                        "/u/www/docs/" )

```

The starting point for all official TCC pages:

```

webstats.py

TCC_WEB_PATH = NMT_WEB_PATH.relative ( "tcc/" )

```

All the files generated by this application are in or under this directory:

```

webstats.py

OUT_WEB_PATH = TCC_WEB_PATH.relative ( "webstats/" )

```

The subdirectories containing the report pages for personal and official users, respectively:

```

webstats.py

PERSONAL_WEB_PATH = OUT_WEB_PATH.relative ( "p/" )
OFFICIAL_WEB_PATH = OUT_WEB_PATH.relative ( "o/" )

```

The suffix to be used on all generated web pages:

```

webstats.py

HTML_EXT = ".html"

```

The generated index page:

```

webstats.py

INDEX_WEB_PATH = OUT_WEB_PATH.relative ( "index" + HTML_EXT )

```

The hit parade page:

```

webstats.py

BY_HITS_WEB_PATH = OUT_WEB_PATH.relative ( "byhits" + HTML_EXT )

```

The prefix for the pages for each unique initial letter of personal account names:

```

webstats.py

LETTER_PREFIX = "pers - "

```

Pathname of the institute homepage relative to the server:

¹⁵ <http://docs.python.org/library/os.path.html>

```
INSTITUTE_HOMEPAGE = '/'
```

6.4. Constants for HTML generation

These constants make it easier to generate some of the common HTML constructs. `TABLE_ATTRS` codifies some universal attributes of the `table` element.

```
TABLE_ATTRS = { 'cellpadding': '3', 'cellspacing': '3',
                'border': '3' }
```

The next two are for specifying alignment in table cells.

```
L_ALIGN = { 'align': 'left' }
R_ALIGN = { 'align': 'right' }
```

`LEAF_NAV_LINKS` is a list of `tccpage2.NavLink` instances used to specify the navigational features on all the pages below the index page. See Section 2.3, “Navigational considerations” (p. 5).

```
LEAF_NAV_LINKS = [
    tccpage2.NavLink ( "Next", [] ),
    tccpage2.NavLink ( "Previous", [] ),
    tccpage2.NavLink ( "TCC Web server statistics",
        [ ("TCC Web server statistics", INDEX_WEB_PATH.url) ] ),
    tccpage2.NavLink ( "Tech Computer Center",
        [ ("Tech Computer Center", TCC_WEB_PATH.url) ] ) ]
```

6.5. Report label text

These constants specify the titles of pages and other text strings.

Title of the index page.

```
INDEX_PAGE_TITLE = "TCC Web server statistics"
```

Title of the hit parade page.

```
BY_HITS_TITLE = "TCC Web server statistics: hit parade"
```

Title of a letter page as a format string, with the NMT URL and the letter to be supplied.

```
LETTER_PAGE_TITLE = 'Personal pages starting with "%s~%s"'
```

Title of an access report page. This is a format string with three values. The first value is the base URL for all pages. The second value is "~" for personal pages, or an empty string for official pages. The third value is the user name or official directory name.

```
ACCESS_REPORT_TITLE = 'Access report for %s%s%s'
```

7. Main program

This program was developing using the Cleanroom or zero-defect software methodology. For more details, see *The Cleanroom software development methodology*¹⁶.

The `webstats.py` program has this overall intended function:

```
webstats.py
# - - - - - w e b s t a t s . p y - - m a i n
def main():
    '''Generate reports on Web page access at this Apache server.

    [ access-logs are readable ->
      index-page := index-content(access-logs)
      hit-parade-page := hits-content(access-logs)
      letter-pages(access-logs) := letter-content(access-logs)
      personal-reports(access-logs) := personal-content(access-logs)
      official-reports(access-logs) := official-content(access-logs)
      sys.stderr += (version-greeting) + (error messages
                    about invalid lines in access-logs, if any) ]
    ...
```

Here are some verification functions that clarify and subdivide the semantics of this intended function. Refer to the specification¹⁷ for an overview of the generated pages; the names just below are just notational shorthand for the parts of the generated structure.

```
webstats.py
#=====
# Verification functions
#-----
# access-logs == Apache access_log file in /var/log/httpd
#   and any files named "access_log.[1-5]" in that directory
#-----
# index-page == a page at INDEX_WEB_PATH
#-----
# index-content(logs) ==
#   (summary of total hits in logs) +
#   (link to hit-parade-page) +
#   (official-report for "/" in logs) +
#   (links to letter-pages(logs)) +
#   (links to official-reports(logs))
#-----
# hit-parade-page == a page at HITS_WEB_PATH
#-----
# hits-content(logs) ==
#   a report showing all pages with at least HITS_CUTOFF hits in
#   logs, in descending order by total hits, with URL as a
#   secondary key
#-----
# letter-pages(logs) ==
#   pages at (LETTER_PREFIX + c + HTML_EXT) relative to
#   OUT_WEB_PATH, one for each unique first letter (c) in the
```

¹⁶ <http://www.nmt.edu/~shipman/soft/clean/>

¹⁷ <http://www.nmt.edu/tcc/projects/tccwebstats4/>

```

# personal page hits from logs
#-----
# letter-content(logs) ==
# for each unique first letter (c) in the personal page hits
# from logs, links to the personal-reports for pages whose
# users start with (c)
#-----
# personal-reports(logs) ==
# pages at (username + HTML_EXT) relative to
# PERSONAL_WEB_PATH for every personal username found in logs
#-----
# personal-content(logs) ==
# one report for each personal account appearing in logs,
# showing all URLs for that account in ascending order by URL
#-----
# official-reports(logs) ==
# pages at (dirname + HTML_EXT) relative to
# OFFICIAL_WEB_PATH for every official directory name found
# in logs
#-----
# official-content(logs) ==
# one report for each official directory appearing in logs,
# showing all URLs for that account in ascending order by URL
#-----

```

First we write a banner message to the standard error stream, and compute the starting and ending time of the report interval. For the definition of the report interval, see Section 6.1, “EXPIRE_DAYS” (p. 7).

webstats.py

```

#-- 1 --
# [ sys.stderr += greeting message and timestamp
# now := the current time as a datetime.datetime
# cutoffTime := a time EXPIRE_DAYS days before the
# current time as a datetime.datetime ]
message ( "==" %s %s == %s +0000\n" %
          ( PRODUCT_NAME, EXTERNAL_VERSION,
            datetime.datetime.utcnow().strftime ( DATE_FORMAT ) ) )
utcZone = FixedTimeZone(0, "UTC")
thirtyDays = datetime.timedelta( EXPIRE_DAYS )
now = datetime.datetime.now(utcZone)
cutoffTime = now - thirtyDays

```

Reading the log files, and pouring their relevant records into the `AccessSummary` instance, are handled in Section 8, “`inputPhase()`: Read the access logs” (p. 13).

webstats.py

```

#-- 2 --
# [ access-logs are readable ->
# accessSummary := a new AccessSummary instance
# containing all the relevant records from those logs ]
accessSummary = inputPhase(cutoffTime, now)

```

Next we create a `tccpage2.TCCPage` instance that will hold the index page. See Section 2.3, “Navigational considerations” (p. 5) for remarks on the navigational features. See also Section 6.3, “Web paths” (p. 8) and Section 6.5, “Report label text” (p. 10).

```

#-- 3 --
# [ indexPage := a new tccpage2.TCCPage instance with a "TCC
#     Computer Center" navigational link ]
navList = [
    tccpage2.NavLink ( "Next", [] ),
    tccpage2.NavLink ( "Previous", [] ),
    tccpage2.NavLink ( "Tech Computer Center",
        [ ("Tech Computer Center", TCC_WEB_PATH.url) ] ) ]
indexPage = tccpage2.TCCPage ( INDEX_PAGE_TITLE, navList,
                               url=INDEX_WEB_PATH.url )

```

The next function adds all content to this page and all subpages.

```

#-- 4 --
# [ indexPage += index-content(accessSummary)
#     hit-parade-page := hits-content(accessSummary)
#     letter-pages(accessSummary) := letter-content(accessSummary)
#     personal-reports(accessSummary) :=
#         personal-content(accessSummary)
#     official-reports(accessSummary) :=
#         official-reports(accessSummary) ]
buildAllPages ( indexPage, accessSummary )

```

Finally, write the content of the index page. For the declaration of `INDEX_ABS_PATH`, see Section 6.3, "Web paths" (p. 8).

```

#-- 5 --
# [ file INDEX_WEB_PATH := indexPage, serialized as XHTML ]
try:
    indexFile = open ( INDEX_WEB_PATH.absPath, "w" )
except IOError, detail:
    fatal ( "Can't open the index page '%s': %s" %
           (INDEX_ABS_PATH.absPath, detail) )
indexPage.write ( indexFile )
indexFile.close()

```

8. inputPhase(): Read the access logs

```

# - - -   i n p u t P h a s e

def inputPhase(cutoffTime, now):
    '''Read all the access logs and return an AccessSummary.

    [ (access-logs are readable) and
      (cutoffTime is the start of the report interval as a
       datetime.datetime) and
      (now is the end of the report interval as a
       datetime.datetime) ->
      return a new AccessSummary containing all the relevant
    '''

```

```
... records from those logs ]
```

We'll always want to read the current access log, whose name is given by `ACCESS_LOG_PATH`. In theory we will also want to read `N_OLDER_LOGS` older, rotated log files, whose names are the same as the current log with extensions `".1"`, `".2"`, and so on. (All these constants are defined in Section 6.2, "Input file paths" (p. 7).) However, the absence of one or all of these older files is not considered an error, because the server may not have been in operation for the full report period.

The first order of business is to create the `AccessSummary` instance that will accumulate the access data, summarized in various ways. See Section 23, "class `AccessSummary`: Principal data structure" (p. 29).

webstats.py

```
## 1 --
# [ accessSummary := an AccessSummary for the interval
#   between cutoffTime and now ]
accessSummary = AccessSummary ( cutoffTime, now )
```

If the current access log doesn't exist or isn't readable, that's a fatal error. There should always be a current access log.

webstats.py

```
## 2 --
# [ if the file at ACCESS_LOG_PATH is readable ->
#   accessSummary := accessSummary with valid records
#   added from that file
#   sys.stderr += messages about lines in that file
#   that aren't valid, if any
# else ->
#   sys.stderr += error message
#   stop execution ]
try:
    readLogFile ( accessSummary, ACCESS_LOG_PATH )
except IOError, detail:
    fatal ( "Can't read the current access log file '%s': %s" %
           (ACCESS_LOG_PATH, detail) )
```

Now add the data from any older logs that exist.

webstats.py

```
## 3 --
# [ if any of the N_OLDER_LOGS files whose names are
#   (ACCESS_LOG_PATH+"."+n), n="1", "2", ... are readable ->
#   accessSummary := accessSummary with valid records
#   added from those files
#   sys.stderr += messages about lines in that file
#   that aren't valid, if any ]
for logNo in range(1, N_OLDER_LOGS + 1):
    logName = "%s.%d" % (ACCESS_LOG_PATH, logNo)
    try:
        readLogFile ( accessSummary, logName )
    except IOError,detail:
        pass

## 4 --
return accessSummary
```

9. readLogFile(): Process one access log

This function reads one access log and adds any valid records to the accessSummary.

webstats.py

```
# - - -   r e a d L o g F i l e

def readLogFile ( accessSummary, fileName ):
    '''Process one access log.

    [ (accessSummary is an AccessSummary instance) and
      (fileName is a string) ->
        if fileName names a readable file ->
            accessSummary := accessSummary with relevant
                records added from that file
            sys.stderr += error messages about invalid lines,
                if any
            else -> raise IOError ]
    ...
    #-- 1 --
    # [ if fileName names a readable file ->
    #     inFile := that file, so opened
    #     else -> raise IOError ]
    inFile = open ( fileName )
```

For the function that turns an access log into a stream of PageGet instances, see Section 51.3, “scanAccessLog(): Scan an access log file” (p. 49).

webstats.py

```
#-- 2 --
# [ accessSummary := accessSummary + (records from inFile
#     that are valid and pass AccessSummary's filters)
#     sys.stderr += error messages about invalid lines
#     from inFile, if any ]
for pageGet in scanAccessLog ( inFile ):
    accessSummary.addPageGet ( pageGet )
inFile.close()
```

10. buildAllPages(): Generate all output pages

webstats.py

```
# - - -   b u i l d A l l P a g e s

def buildAllPages ( indexPage, accessSummary ):
    '''Write all output pages.

    [ (indexPage is a tccpage2.TCCPage instance) and
      (accessSummary is an AccessSummary instance) ->
        indexPage += index-content(accessSummary)
        hit-parade-page := hits-content(accessSummary)
        letter-pages(accessSummary) := letter-content(accessSummary)
        personal-reports(accessSummary) :=
            personal-content(accessSummary)
```

```

        official-reports(accessSummary) :=
            official-reports(accessSummary) ]
    ...

```

All XHTML body content on a `TCCPage` instance is added as children of its `.content` element, which is a `div` element between the header and footer of the page.

The index page has four components: the summary table, the link to the hit parade, the NMT homepage report, and the table of links to personal and official pages. Therefore, we'll parcel this work out to four subfunctions. Two of these subfunctions also build one or more additional pages: the function that builds the link to the hit parade page also builds that page; and the function that builds the personal/official pages table also builds both the single-letter index pages and all of the personal and official reports as well.

See Section 11, “`addSummaryTable()`: Generate the table summarizing all accesses” (p. 17).

webstats.py

```

#-- 1 --
# [ indexPage += (summary of total hits in accessSummary) ]
addSummaryTable ( indexPage.content, accessSummary )

```

See Section 12, “`buildHitParade()`: Build the hit parade” (p. 18).

webstats.py

```

#-- 2 --
# [ indexPage += (link to hit-parade-page made from
#               accessSummary)
#   hit-parade-page := hits-content(accessSummary) ]
buildHitParade ( indexPage.content, accessSummary )

```

See Section 15, “`instituteHomepage()`: Access report for “/”” (p. 21).

webstats.py

```

#-- 3 --
# [ indexPage += (official-content for "/" from accessSummary) ]
instituteHomepage ( indexPage.content, accessSummary )

```

See Section 16, “`buildCategoryTable()`: Build categories table and all personal and official reports” (p. 21).

webstats.py

```

#-- 4 --
# [ indexPage += (links to letter-pages(accessSummary)) +
#               (links to official-reports(accessSummary))
#   letter-pages(accessSummary) := letter-content(accessSummary)
#   personal-reports(accessSummary) :=
#       personal-content(accessSummary)
#   official-reports(accessSummary) :=
#       official-content(accessSummary) ]
buildCategoryTable ( indexPage.content, accessSummary )

```

The last body content on the page is a link to the documentation.

webstats.py

```

#-- 5 --
# [ indexPage += link to the specification ]
indexPage.content.append (
    E.p ( "For documentation describing how these pages "
         "are built, see ",

```

```

        E.a ( "the specification.",
              href="http://www.nmt.edu/tcc/projects/tccwebstats4/" ) )
    )

```

11. addSummaryTable(): Generate the table summarizing all accesses

webstats.py

```

# - - -   a d d S u m m a r y T a b l e

def addSummaryTable ( parent, accessSummary ):
    '''Add the overall summary of accesses.

       [ (parent is an et.Element) and
         (accessSummary is an AccessSummary) ->
         parent += (summary of total hits in accessSummary) ]
    ...

```

We'll use the TABLE_ATTRS dictionary defined in Section 6.4, "Constants for HTML generation" (p. 10) to decorate our table.

webstats.py

```

#-- 1 --
# [ parent += a new 'table' element with attributes TABLE_ATTRS
#   table := that element ]
table = E.table ( TABLE_ATTRS )
parent.append ( table )

```

The first line shows the current time. The second line shows, not the beginning of the report interval, but the time of the oldest actual observed access.

webstats.py

```

#-- 2 --
# [ table += row showing the current time from accessSummary.now ]
timeFormat = "%Y-%m-%dT%H:%M %z"
table.append (
    E.tr (
        E.th ( L_ALIGN, "This report generated" ),
        E.td ( R_ALIGN, accessSummary.now.strftime ( timeFormat ) ) ) )

#-- 3 --
# [ table += row showing start time accessSummary.oldestHit ]
utcZone = FixedTimeZone(0, "UTC")
oldestZulu = accessSummary.oldestHit.astimezone(utcZone)
table.append (
    E.tr (
        E.th ( L_ALIGN, "Accesses since" ),
        E.td ( R_ALIGN, oldestZulu.strftime ( timeFormat ) ) ) )

```

The remaining rows show off-campus, on-campus, and grand total hits.

webstats.py

```

#-- 4 --
# [ table += rows showing total off-campus hits, total

```

```

#      on-campus hits, and total hits from accessSummary ]
nFar = accessSummary.sumHitCount.nFar
nTotal = accessSummary.sumHitCount.nTotal
table.append (
  E.tr (
    E.th ( L_ALIGN, "Requests from off-campus" ),
    E.td ( R_ALIGN, str(nFar) ) ) )
table.append (
  E.tr (
    E.th ( L_ALIGN, "Requests from on-campus" ),
    E.td ( R_ALIGN, str(nTotal-nFar) ) ) )
table.append (
  E.tr (
    E.th ( L_ALIGN, "Total requests" ),
    E.td ( R_ALIGN, str(nTotal) ) ) )

```

12. buildHitParade(): Build the hit parade

webstats.py

```

# - - -   b u i l d H i t P a r a d e

def buildHitParade ( parent, accessSummary ):
    '''Build the hit parade page and the index page link to it.

    [ (parent is an et.Element) and
      (accessSummary is an AccessSummary) ->
        indexPage += (link to hit-parade-page made from
                      accessSummary)
        hit-parade-page := hits-content(accessSummary) ]
    ...

```

The link from the index to the hit parade page is the easy part.

webstats.py

```

#-- 1 --
# [ parent += a new paragraph containing a link to
#   BY_HITS_WEB_PATH ]
parent.append (
  E.p (
    E.a ( "Access report by number of hits",
          href=BY_HITS_WEB_PATH.url ) ) )

```

Next we create a new `TCCPage` instance. For the page title, see Section 6.5, “Report label text” (p. 10). For the definition of `LEAF_NAV_LINKS`, see Section 6.4, “Constants for HTML generation” (p. 10).

webstats.py

```

#-- 2 --
# [ hitsPage := a new tccpage2.TCCPage instance ]
hitsPage = tccpage2.TCCPage ( BY_HITS_TITLE, LEAF_NAV_LINKS,
                              url=BY_HITS_WEB_PATH.url )

```

For the routine that builds the skeleton of an access report table, see Section 13, “accessReport(): Start a new access report table” (p. 19), which returns the `tbody` element under which detail rows will be added.

```

#-- 3 --
# [ hitsPage += a new access report as a 'table' element
#   tbody := the 'tbody' element of that table ]
tbody = accessReport ( hitsPage.content )

```

Now comes the gigantic sort. The method described in Section 39, “AccessSummary.genByHits(): Generate the hit parade” (p. 42) conveniently generates what we need as a stream of HitCount instances in the correct order. All we have to do is make them into access table rows.

```

#-- 4 --
# [ tbody += access report rows made from HitCount
#   instances in accessSummary, sorted according to
#   HitCount.__cmp__ ]
for hitCount in accessSummary.genByHits():
    if hitCount.nTotal < HITS_CUTOFF:
        break
    accessRow ( tbody, hitCount )

```

The page is complete; serialize it to a new file, assuming that we can create the file.

```

#-- 5 --
# [ if BY_HITS_WEB_PATH can be opened new for writing ->
#   that file += hitsPage, serialized as XHTML
#   else ->
#   sys.stderr += error message
#   stop execution ]
try:
    hitsFile = open ( BY_HITS_WEB_PATH.absPath, 'w' )
except IOError, detail:
    fatal ( "Can't open the hit-parade page '%s': %s" %
           (BY_HITS_WEB_PATH.absPath, detail) )
hitsPage.write ( hitsFile )
hitsFile.close()

```

13. accessReport (): Start a new access report table

Access reports occur in several places. Each has the same format: total hits, off-campus percent, and URL.

```

# - - - a c c e s s R e p o r t

def accessReport ( parent ):
    '''Build a new access report

    [ parent is an et.Element ->
      parent += a new access report as a 'table' element
      return the 'tbody' element of that table ]
    ...
#-- 1 --
# [ parent += a new, empty 'tbody' element ]

```

```

# tbody := that element ]
tbody = E.tbody()

#-- 2 --
# [ table := a new 'table' element with three columns,
#     headings for an access report, and tbody as its
#     'tbody' element ]
table = E.table ( TABLE_ATTRS,
                 E.col ( R_ALIGN ),
                 E.col ( R_ALIGN ),
                 E.col ( L_ALIGN ),
                 E.thead (
                     E.th ( "Total" ),
                     E.th ( "Offsite" ),
                     E.th ( "URL" ) ),
                 tbody )
parent.append ( table )

#-- 3 --
return tbody

```

14. accessRow(): Add one row to an access report table

Given a `HitCount` instance, and the `tbody` element of an access report table, this function adds a row in the table displaying that `HitCount`.

webstats.py

```

# - - -   a c c e s s R o w

def accessRow ( tbody, hitCount ):
    '''Add one row to an access table.

    [ (tbody is an et.Element) and
      (hitCount is a HitCount instance) ->
      tbody += an access report row displaying hitCount ]
    ...
#-- 1 --
# [ pct := percentage off-campus from hitCount as a string ]
pct = "%.1f%%" % ( ( 100.0 * hitCount.nFar ) / hitCount.nTotal )

```

Each URL cell in the table is also a link to the page. When converting a URL relative to the server into an absolute URL, we discard the initial "/" because `WebPath.relative()` uses `os.path.join()` to unite the pieces, and if the second piece starts with "/", it will be considered an absolute path, and the first piece will be ignored.

webstats.py

```

#-- 2 --
# [ tbody += a 'tr' element displaying three values:
#     hitCount.nTotal, pct, and a link to hitCount.url ]
pageUrl = NMT_WEB_PATH.relative(hitCount.url[1:]).url
tbody.append (
    E.tr (
        E.td ( R_ALIGN, str(hitCount.nTotal) ),
        E.td ( R_ALIGN, pct ),

```

```
E.td ( L_ALIGN,  
      E.a ( hitCount.url, href=pageUrl ) ) ) )
```

15. instituteHomepage (): Access report for “/”

This is a special one-line access report for URL “/”. It is remotely possible that the homepage will have no hits, so show zero hits in that case. The URL INSTITUTE_HOMEPAGE is defined in Section 6.3, “Web paths” (p. 8).

webstats.py

```
# - - -   i n s t i t u t e H o m e p a g e  
  
def instituteHomepage ( parent, accessSummary ):  
    '''Build the access report for INSTITUTE_HOMEPAGE.  
  
    [ (parent is an et.Element) and  
      (accessSummary is an AccessSummary) ->  
      parent += (official-content for "/" from accessSummary) ]  
    ...  
    #-- 1 --  
    # [ if accessSummary has any access for INSTITUTE_HOMEPAGE ->  
    #   hitCount := its access counts as a HitCount  
    #   else ->  
    #   hitCount := a new HitCount showing zero accesses ]  
    try:  
        hitCount = accessSummary.getUrl ( INSTITUTE_HOMEPAGE )  
    except KeyError:  
        hitCount = HitCount ( INSTITUTE_HOMEPAGE )
```

Building the access report uses Section 13, “accessReport (): Start a new access report table” (p. 19) and Section 14, “accessRow (): Add one row to an access report table” (p. 20).

webstats.py

```
#-- 2 --  
# [ parent += heading "Access report for the NMT homepage" ]  
parent.append ( E.p ( "Access report for the NMT homepage" ) )  
  
#-- 3 --  
# [ parent += a new access report as a 'table'  
#   tbody := the 'tbody' element of that table ]  
tbody = accessReport ( parent )  
  
#-- 4 --  
# [ tbody += an access report row displaying hitCount ]  
accessRow ( tbody, hitCount )
```

16. buildCategoryTable (): Build categories table and all personal and official reports

This function builds the two-column table with links to the reports for personal and official pages.

This is also the logical place to build both the personal letter pages as well all the access report pages, one for each personal account or official directory.

webstats.py

```
# - - -   c a t e g o r y T a b l e

def buildCategoryTable ( parent, accessSummary ):
    '''Build all the personal/official reports and their indices.

    [ (parent is an et.Element) and
      (accessSummary is an AccessSummary) ->
        parent += (links to letter-pages(accessSummary)) +
                  (links to official-reports(accessSummary))
        letter-pages(accessSummary) := letter-content(accessSummary)
        personal-reports(accessSummary) :=
            personal-content(accessSummary)
        official-reports(accessSummary) :=
            official-content(accessSummary) ]
    ...
```

The index table is really only 2 rows \times 2 columns. The first row contains the headings. Each of the two cells of the second row is structured as a vertical stack of unadorned `div` elements.

Thus, the overall flow of this procedure is:

1. Build the table and its first (heading) row.
2. Build a `td` element for the first cell of the second row. Iterate through all the first characters of personal pages using Section 40, “`AccessSummary.genPersonalLetters()`: First letters of personal accounts” (p. 43). For each letter, do all these steps:
 - 2a Add a `div` containing a link to a personal index page for this letter.
 - 2b Create the personal index page for this letter.
 - 2c Iterate through all the personal account names for this letter using Section 41, “`AccessSummary.genPersonals()`: Generate accounts with the same first letter” (p. 43).
For each account name, add a link on the personal index page that points to the personal access report, and also generate that personal access report page.
 - 2d Write the personal index page.
3. Build a `td` element for the second cell of the second row. Iterate through all the official directories using Section 42, “`AccessSummary.genOfficials()`: Generate names of official directories” (p. 44).
For each official directory, add a `div` containing a link to its report page, and also build the report page itself.

webstats.py

```
#-- 1 --
# [ parent += a new 'table' element for the category report
#   personalCell := the 'td' element for personal letter
#     index links
#   officialCell := the 'td' element for official letter
#     index links ]
table = E.table ( TABLE_ATTRS,
                 E.thead (
                   E.tr (
```

```

        E.th ( "Personal pages starting with ",
              E.tt ( "'%s~'" % NMT_WEB_PATH.url ) ),
        E.th ( "Official pages starting with ",
              E.tt ( "'%s'" % NMT_WEB_PATH.url ) ) ) ) )
parent.append ( table )
tbody = et.SubElement ( table, 'tbody' )
tr = et.SubElement ( tbody, 'tr' )
personalCell = et.SubElement ( tr, 'td', valign='top' )
officialCell = et.SubElement ( tr, 'td', valign='top' )

```

For the logic that builds the links to the letter pages, the letter pages themselves, and all the personal access report pages, see Section 17, “`buildPersonalSide()`: Letter and personal pages” (p. 23).

webstats.py

```

#-- 2 --
# [ personalCell += links to letter-pages(accessSummary)
#   letter-pages(accessSummary) := letter-content(accessSummary)
#   personal-reports(accessSummary) :=
#     personal-content(accessSummary) ]
buildPersonalSide ( personalCell, accessSummary )

```

For the logic that builds the links to the official access report pages and the report pages themselves, see Section 21, “`buildOfficialSide()`: Build access reports for official directories” (p. 28).

webstats.py

```

#-- 3 --
# [ officialCell += links to official-reports(accessSummary)
#   official-reports(accessSummary) :=
#     official-content(accessSummary) ]
buildOfficialSide ( officialCell, accessSummary )

```

17. `buildPersonalSide()`: Letter and personal pages

This function builds the index pages by first letter of personal account name, the access reports for each personal accounts, and the links to those pages from the index page.

webstats.py

```

# - - -   b u i l d P e r s o n a l S i d e

def buildPersonalSide ( cell, accessSummary ):
    '''Add links to letter pages, and build letter and personal pages.

    [ (cell is an et.Element) and
      (accessSummary is an AccessSummary instance) ->
        cell += links to letter-pages(accessSummary)
        letter-pages(accessSummary) := letter-content(accessSummary)
        personal-reports(accessSummary) :=
          personal-content(accessSummary) ]
    ...

```

This process is driven by the set of unique first letters of user names. For each letter, we add a link from the index page to the letter page, then build the letter page, then build all the user report pages for that letter. See Section 18, “`buildLetter()`: Build one letter page and related personal pages” (p. 24).

```

#-- 1 --
for letter in accessSummary.genPersonalLetters():
    #-- 1 body --
    # [ cell += link to a letter-page for (letter)
    #   letter-page for (letter) := letter-content for (letter) ]
    #   personal-reports for accounts in accessSummary that start
    #   with (letter) := personal-content for those accounts ]

    #-- 1.1 --
    # [ letterWebPath := WebPath for letter-page for (letter) ]
    relPath = "%s%s%s" % ( LETTER_PREFIX, letter, HTML_EXT )
    letterWebPath = OUT_WEB_PATH.relative ( relPath )

    #-- 1.2 --
    # [ cell += link to letterWebPath.url using link text
    #   (NMT_WEB_PATH + "~" + letter + "...") ]
    linkText = "%s~%s..." % (NMT_WEB_PATH.url, letter)
    cell.append (
        E.div (
            E.a ( linkText, href=letterWebPath.url ) ) )

    #-- 1.3 --
    # [ letter-page for (letter) from accessSummary :=
    #   letter-content for (letter)
    #   personal-reports for accounts in accessSummary that start
    #   with (letter) := personal-content for those accounts ]
    buildLetter ( letter, letterWebPath, accessSummary )

```

18. buildLetter(): Build one letter page and related personal pages

This function builds one of the letter pages, showing links to all personal accounts that start with a given letter. It also builds all the access reports for personal accounts whose names start with that letter.

```

# - - -   b u i l d L e t t e r

def buildLetter ( letter, letterWebPath, accessSummary ):
    '''Build one letter page and all its personal pages.

    [ (letter is a one-character string) and
      (letterWebPath is a WebPath instance) and
      (accessSummary is an Accesssummary instance) ->
        letter-page for (letter) from accessSummary :=
          letter-content for (letter)
        personal-reports for accounts in accessSummary that start
          with (letter) := personal-content for those accounts ]
    ...

```

The first order of business is to start a new `TCCPage` instance to hold the letter page. See Section 6.4, “Constants for HTML generation” (p. 10) and Section 6.5, “Report label text” (p. 10). On that page, the content has only one main element: a bullet list (`ul` element) under which each bullet will be a link.

webstats.py

```
#-- 1 --
# [ letterPage := a new tccpage2.TCCPage instance using
#       the URL from letterWebPath ]
pageTitle = LETTER_PAGE_TITLE % (NMT_WEB_PATH.url, letter)
letterPage = tccpage2.TCCPage ( pageTitle, LEAF_NAV_LINKS,
                               url=letterWebPath.url )

#-- 2 --
# [ letterPage.content += a new 'ul' element ]
ul = et.SubElement ( letterPage.content, 'ul' )
```

To find the complete set of usernames that start with this letter, we'll need the `accessSummary.genPersonals()` method; see Section 23, “class `AccessSummary`: Principal data structure” (p. 29). This method generates all the usernames that start with a given letter, in alphabetical order. See Section 19, “`addPersonalReport()`: Generate links and access report for one personal account” (p. 25).

webstats.py

```
#-- 3 --
# [ ul += 'li' elements containing links to personal-report
#       pages for each personal name in accessSummary
#       that starts with (letter)
#       personal-reports for accounts in accessSummary that start
#       with (letter) := personal-content for those accounts ]
for userName in accessSummary.genPersonals(letter):
    #-- 3 body --
    # [ ul += an 'li' element containing a link to a
    #       personal-report page for (userName)
    #       personal-report for (userName) := personal-content
    #       for (userName) ]
    addPersonalReport ( ul, userName, accessSummary )
```

Serialize the `TCCPage` instance to a file whose name is given by `letterWebPath`.

webstats.py

```
#-- 4 --
# [ file letterWebPath.absPath := letterPage, serialized ]
try:
    pageFile = open ( letterWebPath.absPath, 'w' )
except IOError, detail:
    fatal ( "Can't open letter page '%s': %s." %
           (letterWebPath.absPath, detail) )
letterPage.write ( pageFile )
pageFile.close()
```

19. `addPersonalReport()`: Generate links and access report for one personal account

This function builds the access report page showing all the access statistics for URLs belonging to a given account name. It also adds a link to this page from the parent letter page.

```
# - - -   a d d P e r s o n a l R e p o r t

def addPersonalReport ( ul, userName, accessSummary ):
    '''Build the access report page for one person's account.

    [ (ul is an et.Element) and
      (userName is a personal account name) and
      (accessSummary is an AccessSummary instance) ->
        ul += an 'li' element containing a link to a
              personal-report page for (userName)
        personal-report for (userName) := personal-content
              for (userName) ]
    ...
```

First compute the location of the page in URL and file space. The path is at (userName + HTML_EXT) relative to PERSONAL_WEB_PATH. For these paths, see Section 6.3, “Web paths” (p. 8).

```
#-- 1 --
# [ webPath := a WebPath instance representing userName's
#   page relative to PERSONAL_WEB_PATH ]
fileName = userName + HTML_EXT
webPath = PERSONAL_WEB_PATH.relative ( fileName )
```

Next we build the link from the letter page to the new page.

```
#-- 2 --
# [ ul += an 'li' element containing a link to webPath ]
linkText = "%s~%s" % (NMT_WEB_PATH.url, userName)
ul.append (
    E.li (
        E.a ( linkText, href=webPath.url ) ) )
```

For the actual construction of the user report page, see Section 20, “buildReportPage(): Build one access report page” (p. 26).

```
#-- 3 --
# [ personal-report for (userName) := personal-content
#   for (userName) from accessSummary at webPath ]
buildReportPage ( userName, webPath, "~", accessSummary,
                  accessSummary.genPersonUrls )
```

20. buildReportPage(): Build one access report page

This function builds the access report for a given user or official directory name.

```
# - - -   b u i l d R e p o r t P a g e

def buildReportPage ( userName, webPath, tilde, accessSummary,
                    genUrls ):
    '''Build the access report for one personal web.
```

```

    [ (userName is a user name as a string) and
      (webPath is a WebPath instance) and
      (tilde is "~" for personal pages, "" for official) and
      (accessSummary is an AccessSummary instance) and
      (genUrls is a bound method that generates all the URLs
       for a given user or directory name) ->
        personal-report for (userName) := personal-content
        for (userName) from accessSummary at webPath ]
    ...

```

First we construct a new `TCCPage` instance to hold the report. For `LEAF_NAV_LINKS`, see Section 6.4, “Constants for HTML generation” (p. 10). For `ACCESS_REPORT_TITLE`, see Section 6.5, “Report label text” (p. 10).

webstats.py

```

#-- 1 --
# [ page := a new tccpage2.TCCPage instance with title
#     ACCESS_REPORT_TITLE and navigation list LEAF_NAV_LINKS
#     at path webPath ]
pageTitle = ( ACCESS_REPORT_TITLE %
              (NMT_WEB_PATH.url, tilde, userName) )
page = tccpage2.TCCPage ( pageTitle, LEAF_NAV_LINKS,
                          url=webPath.url )

```

Next we add a new table element to the page that will hold the access report. See Section 13, “`accessReport()`: Start a new access report table” (p. 19).

webstats.py

```

#-- 2 --
# [ page.content += a new access report as a 'table' element
#     tbody := the 'tbody' element of that table ]
tbody = accessReport ( page.content )

```

The `genUrls` argument is a generator that, when passed a user or official directory name, generates all the URLs within that structure. Each URL is then passed to `accessSummary.getUrl()` to obtain a `HitCount` instance enumerating the URL's access count, which is then added to the table body by Section 14, “`accessRow()`: Add one row to an access report table” (p. 20).

webstats.py

```

#-- 3 --
# [ tbody += rows showing the hit counts for URLs
#     generated by genUrls() ]
for url in genUrls(userName):
    #-- 3 body --
    # [ tbody += a row showing the hit counts for URL
    #     from accessSummary ]
    hitCount = accessSummary.getUrl ( url )
    accessRow ( tbody, hitCount )

```

Serialize the completed page to the path specified in `webPath.absPath`.

webstats.py

```

#-- 4 --
# [ file webPath.absPath := page, serialized ]
try:
    pageFile = open ( webPath.absPath, 'w' )

```

```

except IOError, detail:
    fatal ( "Can't open access report page '%s': %s" %
           (webPath.absPath, detail) )
page.write ( pageFile )
pageFile.close()

```

21. buildOfficialSide(): Build access reports for official directories

This function builds all the access report pages for official directories, as well as the links to those pages from the index page.

webstats.py

```

# - - -   b u i l d O f f i c i a l S i d e

def buildOfficialSide ( cell, accessSummary ):
    '''Build all official access reports.

    [ (cell is an et.Element) and
      (accessSummary is an AccessSummary instance) ->
        cell += links to official-reports(accessSummary) ]
      official-reports(accessSummary) :=
        official-content(accessSummary) ]
    ...

```

The generation of official access reports is driven by `accessSummary.genOfficials()`, which generates all the official directory names in ascending order.

webstats.py

```

#-- 1 --
for dirName in accessSummary.genOfficials():
    #-- 1 body --
    # [ cell += link to official-report for dirName
    #   official-report for dirName := official-content
    #     for dirName from accessSummary ]

    #-- 1.1 --
    # [ officialWebPath := WebPath for official-report
    #     for (dirName) ]
    relPath = dirName + HTML_EXT
    officialWebPath = OFFICIAL_WEB_PATH.relative ( relPath )

    #-- 1.2 --
    # [ cell += link to officialWebPath using link text
    #     (NMT_WEB_PATH + dirName + "/") ]
    linkText = "%s%s/..." % (NMT_WEB_PATH.url, dirName)
    cell.append (
        E.div (
            E.a ( linkText, href=officialWebPath.url ) ) )

    #-- 1.3 --
    # [ official-report for dirName := official-content
    #     for dirName from accessSummary ]

```

```
buildReportPage ( dirName, officialWebPath, "",
                  accessSummary, accessSummary.genOfficialUrls )
```

22. fatal (): Write a message and stop

When an any unexpected error occurs, this routine is called to write a message to the standard error stream and terminate execution. Since this script is intended to be run as a cron job, these errors will be e-mailed to the script's owner.

webstats.py

```
# - - -   f a t a l   - - -

def fatal ( *L ):
    """Write an error message and terminate.

    [ L is a list of strings ->
      sys.stderr += elements of L, concatenated
      stop execution ]
    """
    error ( "Fatal error: ", "".join(L) )
    sys.exit ( 1 )
```

23. class AccessSummary: Principal data structure

For a general discussion of what this class's instance holds, and what methods of access are required, see Section 2.1, "Data structures" (p. 3). Here is the interface.

webstats.py

```
# - - - - -   c l a s s   A c c e s s S u m m a r y

class AccessSummary:
    '''Container for all report summary data.

    Exports:
    AccessSummary(cutoffTime, now):
    [ (cutoffTime is the beginning of the report interval
      as a datetime.datetime) and
      (now is the end of the report interval as a
      datetime.datetime) ->
      return a new, empty AccessSummary instance for that
      interval ]
```

The .cutoffTime and .now attributes define the beginning and end of the reporting period. The .oldestHit attribute tracks the oldest timestamp actually observed from a relevant access log entry.

webstats.py

```
.cutoffTime:
    [ the time self.EXPIRE_DAYS in the past as a
      datetime.datetime ]
.now:
    [ the time of instantiation as a datetime.datetime ]
.oldestHit:
```

```
[ the oldest timestamp observed in any access record,  
  as a datetime.datetime instance ]
```

We need to accumulate the total number of hits, and the total of remote hits, for the entire report. We can use a `HitCount` instance to hold those values.

webstats.py

```
.sumHitCount:  
  [ a HitCount instance giving the overall total  
    and remote hit counts for all accesses in self ]
```

The constructor sets up an empty structure. The next method is used to add access records, in the form of `PageGet` instances, to the structure. This method takes care of all the filtering that is done on access records, such as removing records that are too old, not successful, and so forth. See Section 26, “`AccessSummary.addPageGet()`: Process one access record” (p. 33).

webstats.py

```
.addPageGet ( self, pageGet ):  
  [ pageGet is a PageGet instance ->  
    if (pageGet is not older than self.cutoffTime) and  
    (pageGet is relevant by all filtering criteria) ->  
    self := self with that access added ]
```

We'll need a method that retrieves the access counts, represented as `HitCount` instances, for any given URL. This satisfies the need for a way to retrieve the statistics for the NMT homepage, as well as the logic that builds access report pages for specific users and official directories. See Section 38, “`AccessSummary.getUrl()`: Retrieve hit counts for a given URL” (p. 42).

webstats.py

```
.getUrl ( url ):  
  [ url is a URL as a string ->  
    if self has any accesses for url ->  
    return its access counts as a HitCount instance  
    else -> raise KeyError ]
```

To create the hit parade page, we'll need a method that generates a sequence of `HitCount` instances in hit-parade order. See Section 39, “`AccessSummary.genByHits()`: Generate the hit parade” (p. 42).

webstats.py

```
.genByHits():  
  [ generate the HitCount instances in self sorted  
    according to HitCount.__cmp__() ]
```

To create the table of links to pages by their first character, we'll need two methods, one of which generates the first characters of personal pages in order, another to generate all the personal pages that start with that first character. See Section 40, “`AccessSummary.genPersonalLetters()`: First letters of personal accounts” (p. 43) and Section 41, “`AccessSummary.genPersonals()`: Generate accounts with the same first letter” (p. 43).

webstats.py

```
.genPersonalLetters():  
  [ generate the sequence of initial letters of personal  
    account names in ascending order as a sequence of strings ]  
.genPersonals(letter):  
  [ letter is a 1-character string ->  
    generate all the personal account names in self  
    that start with letter ]
```

Next, we'll need a way of retrieving all the official directory names. See Section 42, "AccessSummary.genOfficials(): Generate names of official directories" (p. 44).

webstats.py

```
.genOfficials():
    [ generate all the official directory names in self
      in ascending order ]
```

To generate one personal or official access report page, we'll need to be able to retrieve all the URLs in that account or directory. See Section 43, "AccessSummary.genPersonUrls(): All URLs for a given person" (p. 44) and Section 44, "AccessSummary.genOfficialUrls(): All URLs for an official directory" (p. 45).

webstats.py

```
.genPersonUrls(person):
    [ person is a TCC account name ->
      generate the URLs in self for this person as a
        sequence of strings ]
.genOfficialUrls(dir):
    [ dir is an official directory name ->
      generate the URLs in self for this directory as
        a sequence of strings ]
```

To support these methods, we'll need a number of internal data structures. We'll use the Python `set` type for collections, because most of the time we'll be testing new letters or directories for membership in those collections.

webstats.py

```
State/Invariants:
.__urlMap:
    [ a dictionary whose keys are all the URLs in self
      and each corresponding value is a HitCount instance
        summarizing the hits on that URL in self's reporting
        period ]
.__personalLetterMap:
    [ a dictionary whose keys are the first characters of
      personal directories in self, and each corresponding
        value is a set of the personal directories in self
        that have that first character ]
.__personalMap:
    [ a dictionary whose keys are the names of personal
      directories in self, and each corresponding value
        is a set of the URLs for that person ]
.__officialMap:
    [ a dictionary whose keys are the names of official
      directories in self, and each corresponding value
        is a set of the URLs for that directory ]
...
```

24. Manifest constants for class AccessSummary

Policies for filtering log variables are controlled a number of class variables, declared here.

24.1. AccessSummary . EXPIRE_DAYS: Duration of the report interval

Each report covers the last thirty days.

webstats.py

```
EXPIRE_DAYS = 30
```

24.2. AccessSummary . SYM_DOMAIN: Local domain name, symbolic form

This is a list of strings representing the trailing pair of subdomain and domain names that are considered local to New Mexico Tech.

webstats.py

```
SYM_DOMAIN = ["nmt", "edu"]
```

24.3. AccessSummary . IP_DOMAIN: Local domain in dotted form

This is a list of strings representing the initial elements of an Internet address that is considered local to NMT.

webstats.py

```
IP_DOMAIN = ["129", "138"]
```

24.4. AccessSummary . BAD_STATUS_THRESHOLD: Upper limit for status codes

Status codes (in the `.status` attributes of PageGet instances) that are greater than or equal to this value are not considered successful page accesses.

webstats.py

```
BAD_STATUS_THRESHOLD = 300
```

24.5. AccessSummary . IGNORED_EXTENSIONS: File extensions to be ignored

Accesses to files whose names end with any of these extensions are not considered page loads. Entries in this set should be lowercased, and comparisons against them will be case-insensitive.

webstats.py

```
IGNORED_EXTENSIONS = set ( [ ".bmp", ".css", ".g", ".gif",  
    ".ico", ".jpg", ".jpeg", ".png", ".swf", ".tif", ".tiff" ] )
```

24.6. AccessSummary . SPIDER_STRINGS: Spider detection strings

Access records are considered spider accesses if their `.accessor` field contains any of these strings.

webstats.py

```
SPIDER_STRINGS = ("crawl", "msnbot-", "monitor.nmt.edu")
```

25. AccessSummary.__init__(): Constructor

webstats.py

```
# - - -   A c c e s s S u m m a r y . _ _ i n i t _ _  
  
def __init__ ( self, cutoffTime, now ):  
    '''Constructor for AccessSummary.  
    ...
```

We start by computing the time interval covered by the report. See Section 51.2, “class Fixed-TimeZone” (p. 48) for the time zone identifier; we will use UTC for all times. The `self.oldestHit` attribute is initialized to the current time; as we accept access records, we will compute a running minimum in this value. Then we create the `.sumHitCount` attribute as a new, empty `HitCount` instance—the URL will not be used in this instance.

webstats.py

```
#-- 1 --  
# [ self.cutoffTime := cutoffTime  
#   self.now, self.oldestHit := now  
#   self.sumHitCount := a new, empty HitCount ]  
self.cutoffTime = cutoffTime  
self.now = self.oldestHit = now  
self.sumHitCount = HitCount('')
```

All that remains is to establish the invariants on the various private attributes.

webstats.py

```
#-- 2 --  
self.__urlMap = {}  
self.__personalLetterMap = {}  
self.__personalMap = {}  
self.__officialMap = {}
```

26. AccessSummary.addPageGet(): Process one access record

This method examines an access record, represented by a `PageGet` instance, to see if it is relevant by all our filtering criteria. If relevant, it is added to `self`.

webstats.py

```
# - - -   A c c e s s S u m m a r y . a d d P a g e G e t  
  
def addPageGet ( self, pageGet ):  
    '''Filter and possibly add one access record.  
    ...
```

All the filtering functions are encapsulated in Section 27, “AccessSummary.__isRelevant(): Filter out irrelevant access records” (p. 34). If the record passes the filters, we add it to `self` by calling Section 35, “AccessSummary.__addHit(): Register one access” (p. 38).

webstats.py

```
#-- 1 --  
# [ if pageGet passes all self's filters ->  
#   self := self with pageGet added  
#   else -> I ]
```

```

if self.__isRelevant ( pageGet ) :
    self.__addHit ( pageGet.url, pageGet.when,
        pageGet.isFar ( self.SYM_DOMAIN, self.IP_DOMAIN ) )

```

27. AccessSummary.__isRelevant(): Filter out irrelevant access records

This method examines an access record to see if it passes all the filtering criteria.

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ i s R e l e v a n t

def __isRelevant ( self, pageGet ) :
    '''Filtering for access records.

    [ pageGet is a PageGet instance ->
      if pageGet passes all the filters in
      self.FILTER_FUNCTIONS ->
        return True
      else -> return False ]
    '''

```

This method is essentially the composition of a series of smaller filtering functions, each of which applies one test to a PageGet instance and returns True if it is relevant, False if it should be filtered out.

See Section 34, “AccessSummary.FILTER_FUNCTIONS: Collection of filter functions” (p. 38) for the class variable defines the sequence of these filtering functions.

webstats.py

```

#-- 1 --
# [ if any function in self.FILTER_FUNCTIONS,
#   operating on pageGet, returns False ->
#   return False
#   else -> I ]
for f in self.FILTER_FUNCTIONS:
    if not f(self, pageGet):
        return False

#-- 2 --
return True

```

28. AccessSummary.__statusFilter(): Filter by status code

This method checks a PageGet to see if it represents a successful access. See Section 24.4, “AccessSummary.BAD_STATUS_THRESHOLD: Upper limit for status codes” (p. 32).

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ s t a t u s F i l t e r

def __statusFilter ( self, pageGet ) :
    '''Filter out failed page accesses.

```

```

    [ pageGet is a PageGet instance ->
      if pageGet.status is less than
        self.BAD_STATUS_THRESHOLD ->
        return True
      else -> return False ]
    ...
return pageGet.status < self.BAD_STATUS_THRESHOLD

```

29. AccessSummary.__extFilter(): Ignore certain files by extension

This method checks a PageGet to see if it refers to a file type that is not considered a page fetch, such as image or CSS files. See Section 24.5, “AccessSummary.IGNORED_EXTENSIONS: File extensions to be ignored” (p. 32).

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ e x t F i l t e r

def __extFilter ( self, pageGet ):
    '''Filter out selected file extensions.

    [ pageGet is a PageGet instance ->
      if the file extension of pageGet.url is not
        found in self.IGNORED_EXTENSIONS ->
        return True
      else -> return False ]
    ...
#-- 1 --
# [ ext := file extension from pageGet.url, lowercased ]
front, back = os.path.splitext ( pageGet.url )
ext = back.lower()

#-- 2 --
# [ if ext is in self.IGNORED_EXTENSIONS ->
#   return False
#   else -> return True ]
return ext not in self.IGNORED_EXTENSIONS

```

30. AccessSummary.__spiderFilter(): Filter out search engine spider accesses

This method checks a PageGet to see if it is probably an access by a search engine spider. For the strings that signify spiders, see Section 24.6, “AccessSummary.SPIDER_STRINGS: Spider detection strings” (p. 32). These strings may occur anywhere in the accessor URL.

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ s p i d e r F i l t e r

def __spiderFilter ( self, pageGet ):

```

```

'''Filter out accesses by search engine spiders.

[ pageGet is a PageGet instance ->
  if no string in self.SPIDER_STRINGS is found in
  pageGet.accessor ->
    return True
  else -> return False ]
...
for s in self.SPIDER_STRINGS:
    if s in pageGet.accessor:
        return False
return True

```

31. AccessSummary.__pwdFilter(): Filter out password-protected pages

This method checks a PageGet to see if it is a password-protected page. A PageGet is considered password-protected if its .username field is not a single hyphen.

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ p w d F i l t e r

def __pwdFilter ( self, pageGet ):
    '''Filter out password-protected pages.

    [ pageGet is a PageGet instance ->
      if pageGet pertains to a page that is not
      password-protected ->
        return True
      else -> return False ]
    ...
    return pageGet.username == '-'

```

32. AccessSummary.__timeFilter(): Filter out expired records

This filter rejects PageGet instances that fall before the cutoff time, self.cutoffTime.

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ t i m e F i l t e r

def __timeFilter ( self, pageGet ):
    '''Filter out expired records.

    [ pageGet is a PageGet instance ->
      if pageget.when is in the interval [self.cutoffTime,
      self.now] ->
        return True
      else -> return False ]
    ...
    return self.cutoffTime <= pageGet.when <= self.now

```

33. AccessSummary.__specialFilter(): Special case filter

This filter tests for various minor special cases that tend to crop up in the real world, but are not important to the audience.

```
webstats.py
# - - -   A c c e s s S u m m a r y . _ _ s p e c i a l F i l t e r

def __specialFilter ( self, pageGet ):
    '''Filter out special cases.

    [ pageGet is a PageGet instance ->
      if pageGet is something we would prefer to ignore ->
        return False
      else -> return True ]
    ...
    '''
```

Here are the cases to date that need to be ignored:

- Accesses to the `/robots.txt` file.
- Accesses to URLs starting with `"/~ "`. The server will allow a space after the tilde, but such references are rare.
- If the URL starts with `"/~"` followed by an uppercase letter, the server will redirect that to the lowercased account name. Ignoring them removes bogus entries from the letter table such as `"/K"`.
- For a few access lines, the URL starts with `"http:"`. We can ignore those.

```
webstats.py
#-- 1 --
if pageGet.url.startswith("/robots.txt"):
    return False

#-- 2 --
if pageGet.url.startswith("/~ "):
    return False
```

The next test is for a personal page starting with an uppercase letter. By using the slice expression `[2:3]` instead of just the index expression `[2]`, we don't have to test to insure that the URL is at least three characters long: a slice beyond the end of a string always returns the empty string.

```
webstats.py
#-- 3 --
if pageGet.url[2:3].isupper():
    return False

#-- 4 --
if pageGet.url.startswith("http:"):
    return False
else:
    return True
```

34. AccessSummary.FILTER_FUNCTIONS: Collection of filter functions

The methods contained in this class variable are all applied to access records to see if they are relevant. This sequence is used by Section 27, “AccessSummary.__isRelevant(): Filter out irrelevant access records” (p. 34) to filter access records. These are all unbound methods, so the `self` argument must be provided explicitly when they are called.

webstats.py

```
FILTER_FUNCTIONS = (__statusFilter, __extFilter,
                   __spiderFilter, __pwdFilter, __timeFilter, __specialFilter)
```

35. AccessSummary.__addHit(): Register one access

webstats.py

```
# - - -   A c c e s s S u m m a r y . _ _ a d d H i t

def __addHit ( self, url, when, isFar ):
    '''Register one access.

    [ (url is a URL as a string) and
      (when is the access time as a datetime.datetime) and
      (isFar is True for an off-campus access,
       False for on-campus) ->
      self := self with that access added ]
    ...
```

The first invariant we must establish is the site-total hit count, `self.sumHitCount`. See Section 35, “AccessSummary.__addHit(): Register one access” (p. 38).

webstats.py

```
#-- 1 --
# [ self.sumHitCount += one access with isFar=isFar ]
self.sumHitCount.addHit ( isFar )
```

The next invariant is that `self.oldestHit` is a running minimum of all the access times.

webstats.py

```
#-- 2 --
# [ if pageGet.when < self.oldestHit ->
#   self.oldestHit := pageGet.when
#   else -> I ]
self.oldestHit = min ( self.oldestHit, when )
```

Next we insure that `self.urlMap` contains a `HitCount` instance for this URL, and then we register one hit with that instance. See Section 36, “AccessSummary.__addUrl(): Register one access in self.__urlMap” (p. 39).

webstats.py

```
#-- 3 --
# [ self.__urlMap += one access for url=url and
#   isFar=isFar ]
self.__addUrl ( url, isFar )
```

The remaining invariants are those on the `.__personalLetterMap`, `.__personalMap`, and `.__officialMap` directories. See Section 37, “AccessSummary.__addCategory(): Add URL to appropriate category” (p. 40).

webstats.py

```
#-- 4 --
# [ if url is "/" ->
#     I
#     else if url is for a personal page ->
#         self.__personalLetterMap += entry for the third
#             character of url
#         self.__personalMap += entry for url
#     else ->
#         self.__officialLetterMap += entry for the second
#             character of url
#         self.__officialMap += entry for url ]
self.__addCategory(url)
```

36. AccessSummary.__addUrl(): Register one access in self.__urlMap

webstats.py

```
# - - -   A c c e s s S u m m a r y . _ _ a d d U r l

def __addUrl ( self, url, isFar ):
    '''Update self.__urlMap.

       [ (url is an URL as a string) and
         (isFar is True for off-campus accessor, else False) ->
         self.__urlMap += one access for url=url and
                               isFar=isFar ]

    ...
```

First we make sure that this URL has an entry in `self.__urlMap`. If not, we create a new one. Then, in either case, we register the new access in that `HitCount` instance. See Section 45, “class HitCount: Hit counts for one URL” (p. 45).

webstats.py

```
#-- 1 --
# [ if self.__urlMap does not have url as a key ->
#     self.__urlMap := hitCount := a new HitCount
#         instance for url=url
#     else ->
#         hitCount := self.__urlMap[url] ]
try:
    hitCount = self.__urlMap[url]
except KeyError:
    hitCount = self.__urlMap[url] = HitCount ( url )
```

See Section 47, “HitCount.addHit(): Tally one access” (p. 46).

webstats.py

```
#-- 2 --
# [ if isFar ->
```

```

#     hitCount := hitCount with one total access added
#               and one remote access added
#     else ->
#     hitCount := hitCount with one total access added ]
hitCount.addHit ( isFar )

```

37. AccessSummary.__addCategory(): Add URL to appropriate category

This method maintains the invariants on three dictionaries: `.__personalLetterMap`, `.__personalMap`, and `.__officialMap`. It ignores accesses to the institute homepage `"/`.

webstats.py

```

# - - -   A c c e s s S u m m a r y . _ _ a d d C a t e g o r y

def __addCategory ( self, url ):
    '''Add this URL to the personal or official dictionaries.

    [ url is an URL as a string ->
      if url is "/" or otherwise length 1 ->
        I
      else if url is for a personal page ->
        self.__personalLetterMap += entry for the third
        character of url
        self.__personalMap += entry for url
      else ->
        self.__officialLetterMap += entry for the second
        character of url
        self.__officialMap += entry for url ]
    ...

```

The values in all three of these dictionaries are Python `sets`. They use the `defaultdict` class from the standard Python `collections` module so that we don't have to worry about creating the set on the first access: we can just use the `set` type's `.add()` method. For more information on `defaultdict`, see Section 5, "Imported modules" (p. 6).

The first task is to classify the URL: is it the institute homepage (`"/`), a personal page starting with `"/~`, or an official page starting with `"/` not followed by `~`? See the note at the end of this method regarding a defect in the original version; any URL of length one is ignored here.

webstats.py

```

#-- 1 --
if len(url) < 2:
    return
else:
    maybeTilde = url[1]

```

If the character following the `"/` is a tilde, it's a personal page; otherwise it is an official page.

webstats.py

```

#-- 2 --
# [ if maybeTilde != '~' ->
#     self.__officialMap[directory name from url] += url
#     return

```

```

# else -> I ]
if maybeTilde != '~':
    #-- 2.1 --
    # [ dirName := portion of url[1:] up to the next
    #       "/" if there is one, or to the end otherwise ]
    dirName = url[1:].split('/')[0]

    #-- 2.2 --
    # [ self._officialMap[dirName] += url
    #   return ]
    try:
        self.__officialMap[dirName].add(url)
    except KeyError:
        self.__officialMap[dirName] = set ( [url] )
    return

```

For a personal page, we must add the URL to two sets: one set by first letter, and one set by personal account name.

webstats.py

```

#-- 3 --
# [ dirName := portion of url[2:] up to the next "/" if
#       there is one, or to the end otherwise
#   first := url[2] ]
dirName = url[2:].split('/')[0]
first = url[2]

#-- 4 --
# [ self.__personalLetterMap[first] += dirName
#   self.__personalMap[dirName] += url ]
try:
    self.__personalLetterMap[first].add ( dirName )
except KeyError:
    self.__personalLetterMap[first] = set ( [dirName] )

try:
    self.__personalMap[dirName].add ( url )
except KeyError:
    self.__personalMap[dirName] = set ( [url] )

```

Note

In June 2011, a bogus access log record crashed the script. Here is the original code of step 1, above. Here are the events leading to this failure.

1. The command part of the access log entry looked like this:

```
"GET HTTP/1.1"
```

There were two spaces between GET and HTTP: the URL should have been between those two spaces.

2. In Section 51.11, “scanCmdGroup: Process command group” (p. 62), the command was broken on spaces using `cmdGroup.split(' ')`, and the URL was set to the second element of the result list, which in this case was an empty string.

3. In Section 51.12, “cleanURL(): Process the raw URL” (p. 63), the path part of the URL was normalized by using the standard library’s `os.path.normpath()` function. This has the effect of removing “..” elements from the path, a necessity for security reasons. However, when this function is passed an empty string, it returns ‘.’.
4. Here in this method, I made the assumption that either the URL was just ‘/’ (as the constant `INSTITUTE_HOME PAGE`, defined in Section 6.3, “Web paths” (p. 8)), or it had a second character. Here is the original, flawed code block:

```
#-- 1 --
if url==INSTITUTE_HOME PAGE:
    return
else:
    maybeTilde = url[1]
```

Because the value of `url` at this point was ‘.’, the `else` clause failed with an `IndexError`.

The solution is to ignore here any URL that has a length of one.

38. AccessSummary.getUrl(): Retrieve hit counts for a given URL

webstats.py

```
# - - - A c c e s s S u m m a r y . g e t U r l

def getUrl ( self, url ):
    '''Retrieve the HitCount for a given URL.
    ...
    return self.__urlMap[url]
```

39. AccessSummary.genByHits(): Generate the hit parade

webstats.py

```
# - - - A c c e s s S u m m a r y . g e n B y H i t s

def genByHits ( self ):
    '''Generate the hit parade.
    ...
```

This is the Big Sort: the values in `self.__urlMap` are the complete set of `HitCount` instances. If we sort that set, the `HitCount.__cmp__()` method puts them in the desired order: descending order by hit count, with ascending order by URL the secondary key.

Note that this method does not check for a minimum hit count. The caller should do that, terminating the generator when it starts return entries with too few hits.

webstats.py

```
#-- 1 --
for hitCount in sorted(self.__urlMap.values()):
    yield hitCount

#-- 2 --
raise StopIteration
```

40. AccessSummary.genPersonalLetters(): First letters of personal accounts

Generates the keys in self.__personalLetterMap in ascending order.

webstats.py

```
# - - -   A c c e s s S u m m a r y . g e n P e r s o n a l L e t t e r s

def genPersonalLetters(self):
    '''Generate the first characters of personal accounts.
    ...
    #-- 1 --
    for letter in sorted(self.__personalLetterMap.keys()):
        yield letter

    #-- 2 --
    raise StopIteration
```

41. AccessSummary.genPersonals(): Generate accounts with the same first letter

The first character is used as a key in self.__personalMap; the corresponding value is a Python set of all the personal accounts that start with that character. We generate those account names in ascending order.

webstats.py

```
# - - -   A c c e s s S u m m a r y . g e n P e r s o n a l s

def genPersonals ( self, first ):
    '''Generate all personal account names starting with 'first'.
    ...
    #-- 1 --
    # [ if self.__personalMap has a key (first) ->
    #     nameSet := the corresponding value
    #     else -> raise StopIteration ]
    try:
        nameSet = self.__personalLetterMap[first]
    except KeyError:
        raise StopIteration

    #-- 2 --
    # [ generate the values in nameSet in ascending order ]
    for name in sorted(nameSet):
        yield name

    #-- 3 --
    raise StopIteration
```

42. AccessSummary.genOfficials(): Generate names of official directories

Pretty similar to Section 41, “AccessSummary.genPersonals(): Generate accounts with the same first letter” (p. 43).

webstats.py

```
# - - -   A c c e s s S u m m a r y . g e n O f f i c i a l s

def genOfficials(self):
    '''Generate the official director names, sorted.
    ...
    #-- 1 --
    for dirName in sorted(self.__officialMap.keys()):
        yield dirName

    #-- 2 --
    raise StopIteration
```

43. AccessSummary.genPersonUrls(): All URLs for a given person

This method uses the self.__personalMap dictionary to retrieve the set of URLs. Then we generate that set in sorted order.

webstats.py

```
# - - -   A c c e s s S u m m a r y . g e n P e r s o n U r l s

def genPersonUrls ( self, name ):
    '''Generate all URLs for a personal account.
    ...
    #-- 1 --
    # [ if name is a key in self.__personalMap ->
    #     urlSet := the corresponding value
    #     else -> raise StopIteration ]
    try:
        urlSet = self.__personalMap[name]
    except KeyError:
        raise StopIteration

    #-- 2 --
    # [ generate the members of urlSet in ascending order ]
    for url in sorted(urlSet):
        yield url

    #-- 3 --
    raise StopIteration
```

44. AccessSummary.genOfficialUrls(): All URLs for an official directory

Similar to Section 43, “AccessSummary.genPersonUrls(): All URLs for a given person” (p. 44).

webstats.py

```
# - - -   A c c e s s S u m m a r y . g e n O f f i c i a l U r l s

def genOfficialUrls ( self, name ):
    '''Generate all URLs for a officialal account.
    ...
    #-- 1 --
    # [ if name is a key in self.__officialMap ->
    #     urlSet := the corresponding value
    #     else -> raise StopIteration ]
    try:
        urlSet = self.__officialMap[name]
    except KeyError:
        raise StopIteration

    #-- 2 --
    # [ generate the members of urlSet in ascending order ]
    for url in sorted(urlSet):
        yield url

    #-- 3 --
    raise StopIteration
```

45. class HitCount: Hit counts for one URL

For a discussion of the design of this class, see Section 2.1, “Data structures” (p. 3). We use a new-style class with a `__slots__` attribute to conserve storage, since there will be a lot of instances of this class.

webstats.py

```
# - - - - -   c l a s s   H i t C o u n t

class HitCount(object):
    '''Represents the access data for one URL in the report period.

    Exports:
    HitCount ( url ):
        [ url is a URL as a string ->
          return a new HitCount instance for that URL and
          zero access counts ]
    .url:          [ as passed to constructor, read-only ]
    .nTotal:       [ total hits in self as an int ]
    .nFar:         [ total off-campus hits in self as an int ]
    .addHit ( isFar ):
        [ isFar is True for off-campus, False otherwise ->
          if isFar ->
            self := self with one additional total hit
                   and one additional off-campus hit
```

```

        else ->
            self := self with one additional total hit ]
    .__cmp__(self, other):
        [ if self.nTotal < other.nTotal ->
            return a positive int
          else self.nTotal > other.nTotal ->
            return a negative int
          else ->
            return cmp(self.url, other.url) ]
    ...
    __slots__ = ('nTotal', 'nFar', 'url')

```

46. HitCount.__init__(): Constructor

webstats.py

```

# - - - H i t C o u n t . _ _ i n i t _ _

def __init__(self, url):
    '''Constructor for HitCount.
    ...

    self.url = url
    self.nTotal = self.nFar = 0

```

47. HitCount.addHit(): Tally one access

This method always increments the `.nTotal` attribute. If the URL is considered off-campus, it also increments the `.nFar` attribute.

webstats.py

```

# - - - H i t C o u n t . a d d H i t

def addHit(self, isFar):
    '''Add one hit from the given accessor URL.
    ...

    #-- 1 --
    self.nTotal += 1

    #-- 2 --
    if isFar:
        self.nFar += 1

```

48. Hitcount.__cmp__(): Comparator method

webstats.py

```

# - - - H i t C o u n t . _ _ c m p _ _

def __cmp__(self, other):
    '''Sort descending by .ntotal, ascending by .url
    ...

```

```
return cmp ( (other.nTotal, self.url),
             (self.nTotal, other.url) )
```

49. Hitcount. `__lt__`(): Less-than method

webstats.py

```
# - - -   H i t C o u n t .   _ _   l t   _ _

def __lt__ ( self, other ):
    '''Sort descending by .ntotal, ascending by .url
    ...
    return (other.nTotal, self.url) < (self.nTotal, other.url)
```

50. Epilogue

These lines at the end of the `webstats.py` file cause execution of the main program.

webstats.py

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

51. The `pageget.py` module: Apache log file functions

All the machinery that deals with the Apache access log file (`access_log`) lives in module `pageget.py`.

This module contains two items:

- An instance of class `PageGet` represents one page access. Note that a single access log entry may describe more than one page access.
- Function `scanAccessLog()` is a generator that reads an access log file and generates a stream of `PageGet` objects.

51.1. Prologue to `pageget.py`

Here are the opening declarations to the `pageget.py` module.

pageget.py

```
"""pageget.py: Functions for the Apache access_log recording page fetches.
"""
```

The standard `sys` library gives us access to the standard I/O streams; see <http://docs.python.org/library/sys.html>.

pageget.py

```
#=====
# IMPORTS
#-----
import sys
```

We'll need the `os`¹⁸ module to remove redundant `"/"` and `."` elements from URL paths.

pageget.py

```
import os
```

We use the Python regular expression package `re`¹⁹ to break down the log lines into their component parts.

pageget.py

```
import re
```

The `urllib`²⁰ module provides functions to handle URL encoding and decoding.

pageget.py

```
import urllib
```

The `datetime`²¹ module provides clock and calendar functions, so we can include only records from a given interval.

pageget.py

```
import datetime
```

51.2. class FixedTimeZone

In order to represent aware `datetime` objects (see the `datetime` documentation²² for a discussion of aware vs. naive times), we'll need a class derived from `datetime.tzinfo` to represent the timestamps from the log files.

We don't have to worry about Daylight Saving Time, because the time zone in the log record is encoded as `"-0700"` or `"-0600"` depending on the time of year. So for our uses, a time zone object with a fixed offset will be fine.

The code below was cribbed pretty much directly from the example class `FixedOffset` in the `datetime` documentation, so we'll present it here with minimal comments.

pageget.py

```
# - - - - - c l a s s   F i x e d T i m e Z o n e

class FixedTimeZone(datetime.tzinfo):
    '''Represents a time zone that's always the same offset from UTC.

    Exports:
    FixedTimeZone ( mmEast, name ):
        [ mmEast is a zone offset represented by the number of
          minutes east of UTC, or negative for west) and
          (name is a time zone name as a string) ->
          return a new FixedTimeZone instance with those
          values ]
    .utcoffset(dt):
        [ dt is a datetime.datetime instance ->
          return self's offset as a datetime.timedelta
          instance ]
    .tzname(dt):
```

¹⁸ <http://docs.python.org/library/os.html>

¹⁹ <http://docs.python.org/library/re.html>

²⁰ <http://docs.python.org/library/urllib.html>

²¹ <http://docs.python.org/library/datetime.html>

²² <http://docs.python.org/library/datetime.html>

```

        [ return self's name ]
    .dst(dt):
        [ return a datetime.timedelta of zero ]

    State/Invariants:
    .__offset: [ self's offset as a datetime.timedelta ]
    .__name:   [ self's name, as passed to constructor ]
    .ZERO:    [ a zero datetime.timedelta ]
    ...
ZERO = datetime.timedelta ( 0 )

def __init__ ( self, mmEast, name ):
    '''Constructor
    ...
    self.__offset = datetime.timedelta ( minutes=mmEast )
    self.__name = name

def utcoffset ( self, dt ):
    return self.__offset

def tzname ( self, dt ):
    return self.__name

def dst ( self, dt ):
    return self.ZERO

```

51.3. scanAccessLog (): Scan an access log file

This function processes an access log file, generating a PageGet instance for every access.

pageget.py

```

# - - -   s c a n A c c e s s L o g   - - -

def scanAccessLog ( logFile ):
    """Read an Apache access_log, generating a stream of PageGet objects.

    [ if logFile is a readable file handle ->
      logFile := logFile advanced to end of file
      sys.stderr += messages about lines from logFile that
                  aren't valid access_log lines, if any
      generate a sequence of PageGet objects representing the
                  valid access_log lines from logFile ]
    """

```

There shouldn't be any invalid lines in the access log, but there often are. We'll count them, and if the count is nonzero, we'll send a message to stderr.

pageget.py

```

#-- 1 --
errCount = 0

```

Loop over the lines in the file, generating PageGet objects for accesses from valid lines.

```

#-- 2 --
for rawLine in logFile:
    #-- 2 body --
    # [ if rawLine is a valid access_log line ->
    #     yield a sequence of PageGet objects representing that line
    #     else ->
    #         errCount += 1 ]

```

Because one line in the access log may represent multiple accesses to a page, the `scanAccessLine()` function returns a list of `PageGet` objects, not just a single object.

If the line isn't valid, error logging is done directly to `stderr`, and we get back an empty list. An empty list is considered an error.

```

#-- 2.1 --
# [ if rawLine is a valid access_log line ->
#     getList := a list of one or more PageGet objects
#               representing that line
#     else ->
#     getList := an empty list
#     sys.stderr += error message ]
try:
    getList = scanAccessLine ( rawLine )
except SyntaxError:
    errCount += 1
    getList = []

#-- 2.2 --
# [ generate the elements of getList ]
for get in getList:
    yield get

```

Finally, check for errors and write a summary of the count if there were any.

```

#-- 3 --
# [ if errCount > 0 ->
#     sys.stderr += message about (errCount) errors
#     else -> I ]
if errCount > 0:
    error ( "Count of unrecognizeable access_log lines: %d" %
           errCount )

```

51.4. scanAccessLine(): Process one access log line

This function attempts to process an access log line. Because one line may represent multiple accesses, for valid lines it returns a list of one or more `PageGet` objects representing those accesses.

If the line isn't valid, write an error message to `stderr` and return an empty list.

```

# - - -   s c a n A c c e s s L i n e   - - -

def scanAccessLine ( rawLine ):

```

```

"""Process one access_log line; returns zero or more PageGets.

[ if rawLine is a valid Apache access log line ->
  return a list of zero or more PageGet instances
  representing the accesses described by rawLine
  else ->
    sys.stderr += error message
    raise SyntaxError ]
"""

```

The basic disassembly of the log line into its major parts is done by the `scanGroups()` function.

pageget.py

```

#-- 1 --
# [ if rawLine looks like a valid access_log line at the group level
->
#   accessGroup := contents of accessor group as a string
#   dateGroup  := contents of date group as a string
#   cmdGroup   := contents of command group as a string
#   tailGroup  := contents of tail group as a string
#   else ->
#     sys.stderr += error message
#     return an empty list ]
try:
    (accessGroup, dateGroup, cmdGroup, tailGroup) = scanGroups (
        rawLine )
except ValueError, detail:
    error ( "Group syntax error, %s: %s\n" % (detail, rawLine) )
    raise SyntaxError

```

Then we dispatch each of the four major pieces to its own function to check and process a piece. For the definition of “effective host list,” see the specification²³.

pageget.py

```

#-- 2 --
# [ if accessGroup is a valid host-group ->
#   accessorList := effective host list from accessGroup
#   username := username from accessGroup, or "-" if none
#   else ->
#     sys.stderr += error message
#     return an empty list ]
try:
    accessorList, username = scanAccessGroup ( accessGroup )
except ValueError, result:
    error ( "Host group error, %s: %s" % (result, rawLine) )
    raise SyntaxError

#-- 3 --
# [ if dateGroup is a valid date/time ->
#   when := that date as a datetime.datetime instance
#   else ->
#     sys.stderr += error message
#     return an empty list ]

```

²³ <http://infohost.nmt.edu/tcc/projects/tccwebstats/eff-host-list.html>

```

try:
    when = scanDateGroup ( dateGroup )
except ValueError, result:
    error ( "Date group error, %s: %s" % (result, rawLine) )
    raise SyntaxError

```

This next step removes two special cases. The command may be, instead of GET or POST, some other command such as OPTIONS, HEAD, or PROPFIND. We ignore those cases; they are not page fetches. Also, accesses to URL “//” are treated as accesses to “/”.

pageget.py

```

#-- 4 --
# [ if cmdGroup is a valid command group ->
#     command := command from cmdGroup
#     url      := URL from cmdGroup
# else if the command in cmdGroup is not "GET" or "POST" ->
#     return an empty list
# else ->
#     sys.stderr += error message
#     return an empty list ]
try:
    command, url = scanCmdGroup ( cmdGroup )
    if command not in ("GET", "POST"):
        return []
    if url.startswith ('//'):
        url = url[1:]
except ValueError, result:
    error ( "Command group error, %s: %s" %
           (result, rawLine) )
    raise SyntaxError

#-- 5 --
# [ if tailGroup is a valid tail group ->
#     status := status from tailGroup
# else ->
#     sys.stderr += error message
#     return an empty list ]
try:
    status = scanTailGroup ( tailGroup )
except ValueError, result:
    error ( "Tail group error, %s: %s" % ( result, rawLine ) )
    raise SyntaxError

```

At this point we have parsed the whole record, and `accessorList` contains a list of the accessors. We return a list of `PageGet` objects, one for each element of `accessorList`, using a Python list comprehension.

pageget.py

```

#-- 6 --
# [ return a list of PageGet objects for each accessor in
#     accessorList, using (when, command, url, status)
#     for the other values ]
return [ PageGet ( a, username, when, command, url, status)
         for a in accessorList ]

```

51.5. Breaking down the log line: a mixed-strategy parse

See the specification²⁴ for complete details of the format of the access log file, and where these files live.

In general the access log line can be divided into four groups:

- The *accessor group* describes who is asking for the page.
- The *date group* tells when the request came in.
- The *command group* includes the GET or POST command word, the URL being accessed, and the protocol being used.
- The *tail group* includes the result code, the length of the page fetched (if any), and the referring URL (if known).

A previous version of this program used one large regular expression to break the line into these four groups. The problem with that approach is that the command group is enclosed in double-quotes "...", but occasionally there may be an escaped double quote ("\") inside that string.

It may be possible to use regular expressions to handle this complication, but the author considers that a little too much like rocket science. Instead, the disassembly of the log line proceeds using “mixed strategy parsing:” some of the disassembly uses regular expressions, but some uses more ad-hoc methods. Here’s the general flow:

1. The accessor information consists of four or more blank-delimited fields—if there are multiple accessors, the accessors are separated by one comma and one space.
2. The date group is enclosed within square brackets ([...]). We can use a single regular expression to describe the accessor group as everything up to the "[", and the date group as everything from there up to the "]". This gives us the first two major groups, and what remains is passed on to the next step.
3. What remains after the removal of the date group is the double-quoted string followed by one space and the tail group. This portion is removed by a small routine that scans for an unescaped closing quote. Any escaped quotes found in the string are returned as part of the content.
4. The part remaining after the previous step is the tail section.

These four steps are handled in the next function, `scanGroups()`.

51.6. `scanGroups()`: Top-level disassembly of the log line

As described in the previous section, this function breaks the access log line into its four major parts using a mixed-strategy parse.

The regular expression that breaks off the first two groups is declared globally as a compiled regular expression object `FRONT_RE`. Python’s policy of merging syntactically adjacent string constants is convenient here, allowing us to document each piece of the expression. Python raw strings (`r'...'`) allow us to use backslashes without having to escape them. The constants whose names end with `_GROUP` are symbolic group names used in the regular expression.

pageget.py

```
# - - -   s c a n G r o u p s   - - -  
  
ACCESSOR_GROUP = "a"           # Group ID for accessor group
```

²⁴ <http://www.nmt.edu/tcc/projects/tccwebstats/>

```

DATE_GROUP      = "d"                # Group ID for date group
FRONT_RE       = re.compile (        # Matches first two groups
    r'^'                # Start-of-line anchor
    r'(?P<%s>'          # Start ACCESSOR_GROUP
    r'[\^\\]+'         # Everything up to the next '['
    r')'                # End ACCESSOR_GROUP
    r'\['              # Open bracket for the date group
    r'(?P<%s>'          # Start DATE_GROUP
    r'[\^\\]+'         # Everything up to the next ']'
    r')'                # End DATE_GROUP
    r'\] ' %            # Trailing bracket and space
    (ACCESSOR_GROUP, DATE_GROUP) )

```

Next, we have the actual function.

pageget.py

```

def scanGroups ( rawLine ):
    """Break an access_log line down into its major groups.

    [ if rawLine is a valid access_log line at the group level ->
      return (accessor group, date group, cmd group, tail group)
      as a sequence of strings
    else ->
      raise ValueError ]
    """

```

We start by applying the FRONT_RE regular expression to the entire line, to break off the first two groups. In the match object m, method .group(N) returns the text that matched the group with name N, and method .end() returns the position of the character after the match.

pageget.py

```

#-- 1 --
# [ if rawLine starts with a pattern that matches FRONT_RE ->
#   accessorGroup := group ACCESSOR_GROUP from the match
#   dateGroup     := group DATE_GROUP from the match
#   rest          := rawLine beyond the match
#   else -> raise ValueError ]
m = FRONT_RE.match ( rawLine )
if m is None:
    raise ValueError, "access_log group syntax"
else:
    accessorGroup = m.group ( ACCESSOR_GROUP )
    dateGroup     = m.group ( DATE_GROUP )
    rest          = rawLine[m.end():]

```

Next, the command group is removed using a special function that reads a quote string, allowing for escaped quotes.

pageget.py

```

#-- 2 --
# [ if rest starts with a double-quoted strings, possibly
#   including escaped double-quote characters ->
#   commandGroup := contents of that string (with escaped
#   quotes unescaped)
#   tailGroup    := rest, past that string

```

```
# else -> raise ValueError ]
commandGroup, tailGroup = scanQuoted ( rest )
```

All that remains is to return the sequence of the four group contents.

pageget.py

```
## 3 ##
return (accessorGroup, dateGroup, commandGroup, tailGroup)
```

51.7. scanQuoted (): Process double-quoted string with escapes

The purpose of this function is to recognize a string enclosed in double-quotes, which may contain escaped double-quote characters.

pageget.py

```
# - - - s c a n Q u o t e d - - -
def scanQuoted ( s ):
    """Remove a double-quoted string from the front of s, with escaping

    [ s is a string ->
      if s starts with a double-quoted string with escaping ->
        return (contents of that string with escaped quotes
                unescaped, remainder of s)
      else -> raise ValueError ]
    """
```

We'll use an index `pos` to mark our progress through the string, and a list `L` to accumulate the character content of the string.

pageget.py

```
## 1 ##
pos = 0
L = []
```

`s` must start with a double quote, or we're in trouble right away.

pageget.py

```
## 2 ##
# [ if s starts with '"' ->
#   pos := 1
#   else -> raise ValueError ]
if s[0] == '"':
    pos = 1
else:
    raise ValueError, ( "Expecting '\"' at the start of the "
                        "command group." )
```

Next we move `pos` along until we either find an unescaped quote or run out of string.

A little note on an edge case. In the body of the loop below, we test a string `s[pos:pos+2]` to see if it is an escaped double quote. What happens if `pos` is pointing at the last character of `s`? In that case, the expression `s[pos:pos+2]` evaluates without raising an exception, and returns the single character at `s[pos]`.

```

#-- 3 --
# [ pos := pos advanced to character after closing quote
#       or end of s, whichever comes first
#   L   += characters between s[pos:] and closing quote or
#       end of s, whichever comes first ]
while ( ( pos < len(s) ) and
        ( s[pos] != '"' ) ):
    #-- 3 body --
    # [ if s[pos:] starts with '\"' ->
    #   L   += s[pos+1]
    #   pos += 2
    # else ->
    #   L   += s[pos]
    #   pos += 1 ]
    if ( s[pos:pos+2] == r'\"' ):
        L.append ( s[pos+1] )
        pos += 2
    else:
        L.append ( s[pos] )
        pos += 1

#-- 4 --
# [ if pos < len(s) ->
#   return (elements of L concatenated, s[pos+1:])
# else ->
#   raise ValueError ]
if pos < len(s):
    return ("".join(L), s[pos+1:])
else:
    raise ValueError, ("No closing double-quote: '%s'" % s)

```

51.8. scanAccessGroup(): Process accessors

This function parses the accessor group, consisting of everything up to the first square bracket in the log record.

```

# - - -   s c a n A c c e s s G r o u p   - - -

def scanAccessGroup ( accessGroup ):
    """Determine the set of effective accessor IP addresses from accessGroup

    [ accessGroup is a string ->
      if accessGroup is a valid host-group ->
        return (effective host list from accessGroup,
                username from accessGroup or "-" if none)
      else ->
        sys.stderr += error message
        return an empty list ]

    """

```

The `accessGroup` argument includes a trailing space, so we use the string `.rstrip()` method to remove that. What is left must be four or more fields separated by whitespace:

1. The primary accessor.
2. One or more secondary accessor. If there is more than one, all but the last will have a trailing comma.
3. The penultimate field must be "-".
4. The last field is "-" normally, or the username if the page is password-protected.

pageget.py

```
#-- 1 --
# [ fieldList := fields of accessGroup separated by
#           whitespace, omitting trailing whitespace ]
fieldList = accessGroup.rstrip().split(' ')
```

Note

The original version of the above line had a subtle bug. Originally, the `.split()` call had no argument, so fields were split on clumps of whitespace. Then in the 20050316 log I found this accessor group:

```
'82.115.10.14  - -'
```

Note the two spaces before the first hyphen. That will yield three strings instead of four. The fix is to use `.split(' ')`, which yields the correct four strings.

Next we separate the fields into three groups: the primary accessor in `priHost`; a list of secondary accessors in `secHostList`; and the user name in `username`.

pageget.py

```
#-- 2 --
# [ if fieldList consists of four or more fields of which the
#   next-to-last is "-" ->
#     priHost      := first field of fieldList
#     secHostList  := fields of fieldList from second on,
#                   omitting last two
#     userName     := last field of fieldList
#   else -> raise ValueError ]
if ( ( len ( fieldList ) >= 2 ) and
     ( fieldList[-2] == "-" ) ):
    priHost      = fieldList[0]
    secHostList  = fieldList[1:-2]
    username     = fieldList[-1]
else:
    raise ValueError, ( "Badly formed accessor group: '%s'" %
                       accessGroup )
```

Next we derive the list of effective hosts: if `secHostList` contains only "-", then `priHost` is the only effective host. Otherwise, `secHostList` (with the trailing commas removed from its elements) is the effective host list.

pageget.py

```
#-- 3 --
# [ if secHostList is empty or contains only "-" or "" ->
#   hostList := [ priHost ]
#   else ->
```

```

#     hostList := secHostList with any trailing commas
#                 removed from its elements ]
hostList = findHostList ( priHost, secHostList )

#-- 4 --
if len(hostList) == 0:
    error ( "No hosts found: '%s'" % accessGroup )
return (hostList, username)

```

51.9. findHostList(): Derive the effective host list

This function finds the effective host list, given a primary host and a list of secondary hosts. Here's the interface:

pageget.py

```

# - - -   f i n d H o s t L i s t   - - -
def findHostList ( priHost, secHostList ):
    """Derive the effective host list.

    [ (priHost is the primary host as a string) and
      (secHostList is a list of secondary host names, each
        possibly with a trailing comma) ->
      return the effective host list as a list of strings ]
    """

```

There is a lengthy discussion of the derivation of the effective host list in the specification²⁵. The present function was not in the original design; it was broken out of the `scanAccessGroup()` function because there was just too much logic in that function's third prime.

The first case we eliminate is the case where `secHostList` contains either just "-" or an empty string. In that case, the effective host list contains only the primary host.

pageget.py

```

#-- 1 --
if len(secHostList) == 0:
    return [ priHost ]
else:
    firstSec = secHostList[0]

```

Next we eliminate the cases where the first secondary host is "-" or an empty string. In those cases, we again return a list containing just the primary host.

pageget.py

```

#-- 2 --
if ( ( len ( firstSec ) == 0 ) or
      ( firstSec == "-" ) ):
    return [ priHost ]

```

At this point we have at least one secondary host. The result is this list, with any trailing commas removed from the elements.

²⁵ <http://www.nmt.edu/tcc/projects/tccwebstats/eff-host-list.html>

```

#-- 3 --
hostList = []

#-- 4 --
# [ hostList += a list of the elements of secHostList with any
#           trailing commas removed ]
for secHost in secHostList:
    if secHost.endswith(","):
        hostList.append ( secHost[:-1] )
    else:
        hostList.append ( secHost )

#-- 5 --
return hostList

```

What if one of the elements in `secHostList` is the empty string? The logic above that strips trailing commas is carefully constructed to work with empty strings. The Python expressions `"".endswith(",")` returns `False`, and the expression `""[:-1]` returns the empty string.

51.10. scanDateGroup(): Process date

This function processes the date group of the log line. Its argument is the date group (without its enclosing square brackets); the return value is the date as a `datetime.datetime` instance.

Note

We ignore the zone correction given in the record, because `time.mktime()` expects a local time. This may actually cause a record to be assigned to the wrong day during daylight time transitions. If that is a problem, extract the zone correction along with the other times, convert it to GMT, and then subtract `time.timezone` to re-correct it to local civil time.

We will use a fairly large regular expression to syntax-check the input and break it into its component parts. Here's an example of an actual log file timestamp:

```
08/Feb/2009:04:02:54 -0700
```

Here is the regular expression, annotated:

```

# - - -   s c a n D a t e G r o u p   - - -

DOM_CODE      = "D"           # Day of month field
MON_CODE      = "M"           # Month field (e.g., "Jan")
YYYY_CODE     = "Y"           # Year field
HOUR_CODE     = "h"           # Hour field
MIN_CODE      = "m"           # Minutes field
SEC_CODE      = "s"           # Seconds field
TZSIGN_CODE   = "c"           # Sign of zone correction
TZHH_CODE     = "Z"           # Hours part of zone correction
TZMM_CODE     = "z"           # Minutes part of zone correction

```

```

datePat = re.compile (
    r'(?P<%s>'          # Start DOM_CODE group
    r'\d{2}'           # Day of month
    r')'
    r'/'              # Slash separator
    r'(?P<%s>'        # Start MON_CODE group
    r'[a-zA-Z]{3}'    # Three-letter month code
    r')'
    r'/'              # Slash separator
    r'(?P<%s>'        # Start YYYY_CODE group
    r'\d{4}'           # Four-letter year
    r')'
    r':'              # Colon separator
    r'(?P<%s>'        # Start HOUR_CODE group
    r'\d{2}'           # Hour
    r')'
    r':'              # Colon separator
    r'(?P<%s>'        # Start MIN_CODE group
    r'\d{2}'           # Minute
    r')'
    r':'              # Colon separator
    r'(?P<%s>'        # Start SEC_CODE group
    r'\d{2}'           # Second
    r')'
    r' '              # Matches one space
    r'(?P<%s>'        # Start TZSIGN_CODE group
    r'[\-+]'          # Matches '+' or '-'
    r')'
    r'(?P<%s>'        # Start TZHH_CODE group
    r'\d{2}'           # Two digits
    r')'
    r'(?P<%s>'        # Start TZMM_CODE group
    r'\d{2}'           # Two digits
    r')'
    % ( DOM_CODE, MON_CODE, YYYY_CODE, HOUR_CODE, MIN_CODE,
        SEC_CODE, TZSIGN_CODE, TZHH_CODE, TZMM_CODE ) )

```

We'll also need a dictionary to translate month names into month numbers:

pageget.py

```

monthDict = { "jan": 1, "feb": 2, "mar": 3, "apr": 4,
              "may": 5, "jun": 6, "jul": 7, "aug": 8,
              "sep": 9, "oct": 10, "nov": 11, "dec": 12 }

```

The function header and intended function:

pageget.py

```

def scanDateGroup ( dateGroup ):
    """Extract the access timestamp from the raw dateGroup

    [ dateGroup is a string ->
      if dateGroup is a valid date/time ->
        return that date as an epoch time
      else ->

```

```
        raise ValueError ]
    """
```

First we apply the regular expression to the argument, and then extract the subfields into six local variables. If the match fails, we consider the line badly formed.

pageget.py

```
#-- 1 --
# [ if dateGroup matches datePat ->
#     m := a Match object for that match
#     else ->
#         raise ValueError ]
m = datePat.match ( dateGroup )
if m is None:
    raise ValueError, "Bad date format: '%s'" % dateGroup

#-- 2 --
yyyy = int ( m.group ( YYYY_CODE ) )
mon  = m.group ( MON_CODE )
dom  = int ( m.group ( DOM_CODE ) )
hh   = int ( m.group ( HOUR_CODE ) )
mm   = int ( m.group ( MIN_CODE ) )
ss   = int ( m.group ( SEC_CODE ) )
rawSign = m.group ( TZSIGN_CODE )
tzhh  = int ( m.group ( TZHH_CODE ) )
tzmm  = int ( m.group ( TZMM_CODE ) )
```

Next, translate the month name into a month number using the `monthDict` mapping.

pageget.py

```
#-- 3 --
# [ if (mon is a valid three-character month code) ->
#     monthNo := the corresponding month number
#     else -> raise ValueError ]
try:
    monthNo = monthDict [ mon.lower() ]
except:
    raise ValueError, ( "Unknown month code '%s'" % mon.lower() )
```

The zone correction for our `FixedTimeZone` class (see Section 51.2, “class `FixedTimeZone`” (p. 48)) is expressed in minutes. Combine the sign (`rawSign`), hours (`tzhh`), and minutes (`tzmm`). Since this application will probably never run anywhere but Mountain Time, you may think that this kind of generality is overkill, but who will be laughing next time the earth's poles shift?

pageget.py

```
#-- 4 --
# [ if rawSign is "-" ->
#     zone := a FixedTimeZone instance representing
#             (tzhh) hours and (tzmm) minutes west of UTC
#     else ->
#     zone := a FixedTimeZone instance representing
#             (tzhh) hours and (tzmm) minutes east of UTC ]
zoneName = "%s%02d%02d" % (rawSign, tzhh, tzmm)
if rawSign == "-":
    mmEast = - ( tzhh * 60 + tzmm )
else:
```

```
mmEast = tzhh * 60 + tzmm
zone = FixedTimeZone ( mmEast, zoneName )
```

Now we have all the pieces we need to assemble a zone-aware `datetime.datetime` instance. The seventh argument is microseconds; the eighth is the zone correction.

pageget.py

```
## 5 --
return datetime.datetime ( yyyy, monthNo, dom, hh, mm, ss, 0, zone )
```

51.11. scanCmdGroup: Process command group

Of the “command group,” the only part we care about is the command (typically GET or POST) and the URL.

In the vast majority of log lines, there are three parts separated by spaces: the command, the URL, and the protocol. However, there are two infrequent cases that differ:

- Sometimes the protocol group may be missing.
- The URL may have embedded, unescaped spaces in it. In that case, we need to preserve those spaces. There may be multiple URLs that differ only past the first space.

Hence, our approach does the right thing in those cases:

1. First we break the URL up on space characters. This gives us a list of at least two elements.
2. If the last element of this list starts with the string "HTTP", we discard it.
3. The first element of the resulting list is the command. The remaining elements are reassembled with the intervening blanks to form the URL.

pageget.py

```
# - - -   s c a n C m d G r o u p   - - -
def scanCmdGroup ( cmdGroup ):
    """Extract the command and URL from the command group

    [ if cmdGroup is a string ->
      if cmdGroup is a valid command group ->
        return (command from cmdGroup, URL from cmdGroup)
      else ->
        raise ValueError ]
    """
```

First we use `.split()` to break on spaces.

pageget.py

```
## 1 --
# [ wordList := cmdGroup broken up on space characters ]
wordList = cmdGroup.split ( ' ' )
```

Next, we remove the protocol element from the end, if there is one.

pageget.py

```
## 2 --
# [ if wordList[-1] starts with "HTTP" ->
```

```

#     wordList := wordList without its last element
#     else -> I ]
if wordList[-1].startswith ( "HTTP" ):
    wordList.pop()

```

Finally, we reassemble the URL part and return a tuple containing the command and the URL. We use Section 51.12, “cleanURL(): Process the raw URL” (p. 63) to remove URL-encoding and do other cleanup tasks.

Note

In January 2011, some log entries turned up that had null bytes in them. This crashed the script because the `etbuilder.py` module translates strings to Unicode, and null bytes cannot be Unicode-encoded. Hence the test below that discards such entries.

In June 2011, some log entries came through that had command group “GET HTTP/1.1”, with two spaces after GET—the URL was completely missing. Hence the test for that below as well.

pageget.py

```

#-- 3 --
# [ if url is empty ->
#     raise ValueError
#     else ->
#     decodedURL := url, with URL-encoding decoded, minus any
#                 "?..." tail ]
url = " ".join ( wordList[1:] )
if len(url) == 0:
    raise ValueError("The URL is missing: '%s'" % cmdGroup)
else:
    decodedURL = cleanURL(url)

#-- 3 --
# [ if decodedURL contains any null characters ->
#     raise ValueError
#     else ->
#     return (first element of wordList, decodedURL) ]
if '\x00' in decodedURL:
    raise ValueError("Nulls disallowed: %r" % url)
else:
    return (wordList[0], decodedURL)

```

51.12. cleanURL(): Process the raw URL

This function takes the URL from the log, reverses URL encoding, and removes any CGI arguments.

Although Python's `urllib` module has a perfectly reasonable function to decode URL encoding, in December 2005, an earlier version of this function led to the script crashing. Here is the story of that crash and the resulting fix.

A certain user, no doubt in Windows, created a URL containing the string “Tönen”. The “ö” character is represented as “\xf6”. That character was placed into an XHTML page, built using the Document Object Model (DOM).

Unfortunately, the DOM expects all strings to be encoded using UTF-8. Character code "\xf6" is not valid in UTF-8 strings, so the DOM serializer failed when it tried to convert the string to Unicode using the Python function "unicode(text, "utf-8")". (For more information about Unicode, see *The Unicode Standard, Version 4.0*²⁶.)

We could fix this by leaving the URL encoded, but that escapes a number of quite reasonable characters, including "~", so that all URLs of user pages would start "%7Eusername/...".

A better fix is to go through the characters of the URL and apply URL-encoding *only* to characters whose codes are not ASCII, that is, whose codes are 0x80 or greater. One advantage of this is that users can paste URLs containing such characters into the browser, and they'll get the correct URL.

pageget.py

```
# - - -   c l e a n U R L   - - -

def cleanURL ( rawURL ):
    """Remove from an URL any URL-encoding and "?..." tail

    [ if rawURL is a string ->
      return rawURL with URL encoding decoded and minus
      any CGI arguments ]
    """
```

First we remove the CGI arguments.

pageget.py

```
#-- 1 --
# [ if rawURL contains any "?" characters ->
#   head := rawURL up to the first "?" character
#   else ->
#     head := rawURL ]
L       = rawURL.split ( "?" )   # Discard up to first "?"
head    = L[0]
```

Reversing URL decoding is handled by the Python `urllib` module²⁷'s `unquote()` function.

pageget.py

```
#-- 2 --
# [ unquoted := head with URL-encoded characters unquoted ]
unquoted = urllib.unquote ( head )
```

The next line removes redundant "/" and ".." elements from the URL, then requotes all non-ASCII characters. See Section 51.13, "asciifyString: Encode non-ASCII characters" (p. 64).

pageget.py

```
#-- 3 --
# [ return unquoted with redundant "/" and ".." groups
#   removed and all characters with codes >= 0x80
#   replaced by their URL-encoded forms ]
return os.path.normpath ( asciifyString ( unquoted ) )
```

51.13. asciifyString: Encode non-ASCII characters

This function replaces non-ASCII characters in a string with their URL-encoded equivalents. For an explanation of why we need to do this, see Section 51.12, "cleanURL(): Process the raw URL" (p. 63).

²⁶ <http://www.unicode.org/versions/Unicode4.0.0/>

²⁷ <http://docs.python.org/lib/module-urllib.html>

```
# - - -   a s c i i f y S t r i n g   - - -

def asciifyString ( s ):
    """Like urllib.quote(), but only quotes non-ASCII characters.

       [ s is a string ->
         return s with all characters >= 0x80 escaped using
         URL encoding ]
    """
```

URL encoding replaces characters with the string "%XX", where XX is the character's hex code. For the function that does the escaping, see Section 51.14, "asciifyChar(): Escape a non-ASCII character" (p. 65).

```
#-- 1 --
# [ cleaned := a list of the characters from s, with those
#           >= 0x80 replaced by the URL-encoded equivalent ]
cleaned = [ asciifyChar(c) for c in list(s) ]

#-- 2 --
return "".join ( cleaned )
```

51.14. asciifyChar(): Escape a non-ASCII character

Given a character, this function returns the character if it is ASCII, or the URL-encoded equivalent if it is not ASCII. Since ASCII is a seven-bit code, we consider characters with codes 0x80 and greater to be non-ASCII.

```
# - - -   a s c i i f y C h a r   - - -

def asciifyChar ( c ):
    """Escape c if it isn't ASCII, otherwise return c.

       [ c is a one-character string ->
         if ord(c) < 0x80 ->
           return c
         else ->
           return '%XX' where XX is ord(c) ]
    """
    if ord(c) < 0x80:
        return c
    else:
        return "%%%2X" % ord(c)
```

51.15. scanTailGroup(): Process remaining fields

Three fields remain at the end of the access log record: the result code (e.g., 404 for "Page not found"), the length of the returned file, and the referring URL enclosed in double quotes. An earlier version of this module extracted the referrer, and that was a field in the PageGet object, but this version will not retain that field.

So all we need is a pattern that matches an integer followed by one space.

pageget.py

```
# - - -   s c a n T a i l G r o u p   - - -

STATUS_FIELD    = "c"           # Result code, e.g., 200 ok, 404 not found

tailPat = re.compile (
    r' '           # Space before status code
    r'(?P<%s>'    # Start STATUS_FIELD group
    r'\d+'        # Result code: all leading digits
    r')'          # One space
    % STATUS_FIELD )

def scanTailGroup ( tailGroup ):
    """Extract the status from the tail group.

    [ if tailGroup is a valid tail group ->
      return status from tailGroup
      else -> raise ValueError ]
    """
```

First we apply the regular expression to the raw tail. Then, assuming it matches, we extract the status code, convert it to an integer, and return it as the result.

pageget.py

```
#-- 1 --
# [ if tailGroup matches tailPat ->
#   m := a Match object describing that match
#   else ->
#     raise ValueError ]
m = tailPat.match ( tailGroup )
if m is None:
    raise ValueError, ( "Invalid status/length/referrer group '%s'" %
                        tailGroup )

#-- 2 --
# [ return STATUS_FIELD from m as an integer ]
status = int ( m.group ( STATUS_FIELD ) )
return status
```

51.16. class PageGet: Describes one page access

The PageGet object is an abstract data type containing the relevant information from the access log about one page access attempt.

Note that one access log line may be translated into zero, one, or several PageGet objects.

Here is the exported interface for this class:

pageget.py

```
# - - - - -   c l a s s   P a g e G e t   - - - - -
```

```

class PageGet:
    """Class to represent one line from the Apache access_log

    Exports:
    PageGet ( accessor, username, when, command, url, status ):
        [ if (accessor is the remote site's IP address as a string)
          and (username is the username if the page is under
            password protection, "-" otherwise)
          and (when is the time of the access as a datetime.datetime)
          and (command is the command, e.g., "GET" or "POST")
          and (url is the URL the accessor wants to load)
          and (status is the usual HTTP return status as an integer) ->
            return a new PageGet object representing that access ]
    .accessor:      [ as passed to constructor ]
    .username:      [ as passed to constructor ]
    .when:          [ as passed to constructor ]
    .command:       [ as passed to constructor ]
    .url:           [ as passed to constructor ]
    .status:        [ as passed to constructor ]
    .isFar ( symDomain, ipDomain ):
        [ if (symDomain is a list of strings representing the
          symbolic form of the local domain such as ["nmt","edu"])
          and (ipDomain is a list of strings representing the
            numeric form of the local domain such as ["129","138"]) ->
            if (self.accessor is a single name)
              or (its last parts match symDomain)
              or (its first parts match ipDomain) ->
                return False
            else -> return True ]
    str(self):
        [ return a string representing self for debug use ]
    """

```

51.17. PageGet.__init__(): Constructor

The constructor just saves its arguments.

pageget.py

```

# - - -   P a g e G e t . _ _ i n i t _ _   - - -

def __init__ ( self, accessor, username, when, command, url, status ):

    """Constructor for PageGet"""
    self.accessor = accessor
    self.username = username
    self.when     = when
    self.command  = command
    self.url      = url
    self.status   = status

```

51.18. PageGet.isFar(): Is this an off-campus accessor?

This function tests to see whether an accessor's site is considered off-campus or not.

We expect that the accessor will be either a single name, or a form like "a.b.c.d". An accessor is considered local in the former case, or if its trailing parts match `symDomain`, or if its leading parts match `ipDomain`.

pageget.py

```
# - - - P a g e G e t . i s F a r - - -

def isFar ( self, symDomain, ipDomain ):
    """Is this record from our domain?

        [ (symDomain is a list of the trailing parts of the local IP
          domain as strings) and
          (ipDomain is a list of the leading parts of the local
          IP domain as numbers in string form) ->
          if self represents a fetch from symDomain or
          ipDomain ->
            return False
          else -> return True ]
    """
```

First we split the accessor's domain using "." as the delimiter:

pageget.py

```
#-- 1 --
# [ L := list of parts of self.accessor split on "." ]
L = self.accessor.split ( "." )
```

Before trying to match pieces of `L` against `symDomain` and `ipDomain`, we need to make sure that `L` is at least as long as they are. If `L` is too short, there are two cases:

1. If the accessor is recorded as "unknown", that's considered a remote accessor.
2. Otherwise this is considered a local access, because accesses from local machines omit the ".nmt.edu" qualifier.

pageget.py

```
#-- 2 --
# [ if L is at least as long as both symDomain and
#   ipDomain ->
#     I
#   else if L[0] is "unknown" ->
#     return False
#   else -> return True ]
if ( ( len ( L ) < len ( symDomain ) ) or
      ( len ( L ) < len ( ipDomain ) ) ):
    if L[0] == "unknown": return True
    else:                  return False
```

Now that we know `L` is at least as long as `symDomain` and `ipDomain`, we can compare its leading parts against `ipDomain` and its trailing parts against `symDomain`.

pageget.py

```
#-- 3 --
# [ if ipDomain matches the initial elements of L ->
```

```

#     return False
#     else -> I ]
if ipDomain == L[:len(ipDomain)]:
    return False

#-- 4 --
# [ if symDomain matches the final elements of L ->
#     return False
#     else ->
#     return True ]
if symDomain == L[-len(symDomain):]: return False
else:                                return True

```

51.19. PageGet.__str__(): Debug display

This class is provided with a `.__str__()` function for debug display of PageGet objects.

pageget.py

```

# - - -   P a g e G e t . _ _ s t r _ _   - - -

TIME_FORMAT = "%Y-%m-%dT%H:%M:%S %Z"

def __str__( self ):
    "Output self as a string"
    humanDate = self.when.strftime ( self.TIME_FORMAT )

    if self.username != "-":
        isPassword = " [PWD]"
    else:
        isPassword = ""

    return ( "%s %s%s %s %s %s" %
            ( humanDate, self.accessor, isPassword,
              self.command, self.status, self.url ) )

```

51.20. error(): Write a message to stderr

This function writes a message to the standard error stream. It takes a list of strings as arguments, concatenates them, and sends them to `sys.stderr` decorated with a characteristic prefix so we can rapidly find error messages in log files. It also appends a newline, so none of the callers have to remember to do that. The message is also written to file `ERROR_LOG` with append access so there is a cumulative log of all errors.

pageget.py

```

# - - -   e r r o r   - - -

ERROR_LOG = "/u/www/docs/tcc/webstats/error-log"

def error ( *L ):
    """"Send a message to the standard error stream.

    [ L is a list of strings ->
      sys.stderr += (error message prefix) +

```

```

        (elements of L, concatenated) + "\n" ]
    """
    text = "*** %s\n" % "".join(L)
    message ( text )

```

51.21. message () : Send a message to standard error and log file

This function sends a string to the standard error stream, and also appends it to a cumulative log file.

pageget.py

```

# - - -   m e s s a g e   - - -

def message ( text ):
    """Send a message to stderr and ERROR_LOG.

    [ text is a string ->
      sys.stderr += text
      file ERROR_LOG += text ]
    """
    sys.stderr.write ( text )
    logFile = open ( ERROR_LOG, "a" )
    logFile.write ( text )
    logFile.close()

```

51.22. A small test driver for PageGet

Here is a small test driver program for the PageGet class. It reads from standard input a file in `access_log` format and displays each record's output on standard output.

pagegettest

```

#!/usr/bin/env python
#--
# pagegettest: Test driver for pageget.py
# For documentation, see www.nmt.edu/tcc/projects/webstats/ims/
#--
# Reads from standard input, expects a file that looks like access_log.
#--
import string, sys, datetime
from pageget import *

SYM_DOMAIN = ["nmt", "edu"]
IP_DOMAIN = ["129", "138"]

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

for pageGet in scanAccessLog ( sys.stdin ):
    if pageGet.isFar ( SYM_DOMAIN, IP_DOMAIN ): local = "R"
    else: local = "L"

    print ( "%s %d %s %s\n          %s" %
            (local, pageGet.status,

```

```
pageGet.when.strftime ( "%Y-%m-%d %H:%M:%S %z" ),  
pageGet.accessor, pageGet.url )
```

