⟨*copyright*⟩≡

```
#!/usr/bin/env python2

# Copyright (c) 2004 CollabNet.  All rights reserved.

##
## Publish ClearCase from/to SVN.  Run 'sync.py --help' for details.
```

⟨*def print general usage*⟩≡

```
def print_general_usage():
    usage = """ \nUsage: clearsvn.py [-h] | import <args>... | export <args>...

Description:

   This  script  is used  to  synchronize  a  ClearCase repository  with  a
   SubVersion  (SVN) repository,  preserving the  history and  integrity of
   changes to directories  as much as possible. You can  run this script in
   either import mode, to pull changes  in to ClearCase from SVN; or export
   mode to push changes from ClearCase to SVN.

   For more information about how to invoke an import or export, do either:

   clearsvn.py import -h, or,
   clearsvn.py export -h

 """
    print usage
```

⟨*def print export usage*⟩≡

```
def print_export_usage():
    usage = """ \nUsage: clearsvn.py export [-h] |
        [-v <mode>] [-n] [-f] -s <path> -c <path> -b <path> [-l <path>]

Description:

   In export mode, this script seeks  to synchronize the state of files and
   directories from ClearCase into a SubVersion workspace. Migration out of
   ClearCase is  view based; it is up  to you what branch  or branches your
   view is configured for.

   You specify  two paths  (including the VOB)  within a \'current\'  and a
   \'before\'  view.  This script  compares the  directory trees  rooted at
   both paths  to detect  the changes to  migrate.  The script  also checks
   that the contents under the \'before\' path exactly match the SubVersion
   workspace, for  if not,  then applying the  delta between the  two views
   will not  produce PREDICTABLE  results.  You have  to use  -f/--force to
   cause the script to proceed in that case.

   The -n option provides for doing test runs with no actual updates.

   You should  validate and  commit all such  migrated changes at  your own
   discretion after the script has completed.

   Unless a fatal  error should cause the script to abort,  you will see an
   error and warning summary printed at the end of the run.

Command Line Options:

  -h | --help                 : display this usage information
  -f | --force                : proceed even when the before view and the
                              :   svn client do not match. (not recommended)
  -s <path> | --svn-root      : tells where the SVN client root is located
  -c <path> | --ccview-root   : tells the path, as described above, into
                              :   a ClearCase view and VOB for
                              :   the \'current\' configuration.
  -b <path> | --ccref-view-root  : tells the path, as described above,
                                 :  to the \'before\' configuration.
  -v <mode> | --verbosity     : verbosity: (\"low\",\"med\", or \"high\").
                              :   Defaults to low.
  -l <path> | --log-file      : tells the path and filename of a logfile.
                              :   Defaults to ./log.txt (in the current dir).
  -n                          : no-op. Provides for running the script,
                              :   finding directory changes and comparing
                              :   files, but without making any changes in
```

```
                                :   any repository. File comparisons may be wrong
                                :   due to directory changes (like adds) not
                                :   having been made.

   Arguments may be given in any order.

   """
      print usage
```

⟨*def print import usage*⟩≡

```
def print_import_usage():
    usage = """ \nUsage: clearsvn.py import [-h] |
        [-v <mode>] [-n] -s <path> -c <path> -i <path> -t <path> -r n:m

Description:

   In import mode, this script seeks to synchronize changes from a
   SubVersion workspace into a ClearCase view.

   Directory changes are synchronized first. This script runs an svn log
   command in verbose mode to discover the net directory changes (file
   adds, moves, and deletes) in the specified revision range. The script
   applies the directory updates in ClearCase. Next, the script does a
   tree walk over the files and directories in the specified folder under
   the ClearCase view comparing with the contents found in the SubVersion
   workspace. Versioned files that are found to differ are checked out and
   updated in the ClearCase view.

   The -n option provides for doing test runs with no actual updates.

   You should validate and checkin all such migrated changes at your own
   discretion after the script has completed.

   Unless a fatal error should cause the script to abort, you will see an
   error and warning summary printed at the end of the run.

Command Line Options:

  -h | --help                : display this usage information
  -s <path> | --svn-root     : tells where the SVN client root is located
  -c <path> | --ccview-root  : tells a path within a view and VOB in
                             :   ClearCase. This is the folder to where
                             :   SubVersion changes are imported.
  -t <path> | --tree-path    : tells what part of the tree was checked out
                             :   of the repository.
  -i <path> | --inc-path     : tells what part of the tree to include.
                             :   All other parts of the tree are ignored.
  -v <mode> | --verbosity    : verbosity: (\"low\",\"med\", or \"high\").
                             :   Defaults to low.
  -r n:m                     : the range flag whose argument is passed
                             :   through to the svn log --verbose command
  -l <path> | --log-file     : tells the path and filename of a logfile.
                             :   Defaults to ./log.txt (in the current dir).
  -n                         : no-op. Provides for running the script,
                             :   finding directory changes and comparing
```

```
                                :   files, but without making any changes in
                                :   any repository. File comparisons may be wrong
                                :   due to directory changes (like adds) not
                                :   having been made.

   Arguments may be given in any order.

   Example:

    clearsvn.py import -s D:\svnstuff\client3\our_project ^
                -c M:\Bob_view\\the_vob\our_project ^
                -i \\trunk -t \\trunk ^
                -v med ^
                -r 217:213

  (where the ^ characters are the DOS command line continuation)
   """
      print usage
```

This is meant to be a test chunk to show user input being accepted by a Python script. We do that using `sys.stdin.readline()`.

Next, what we want to do is show that we can have a user enter a digit to select from a set of choices.

⟨*test sync*⟩≡

```
import sys,string,re
CmdError="error"

⟨def get choice index⟩
choices=['1st','2nd','3rd','4th']
get_choice_index(choices,4)
```

What we want this routine to do is take an optional argument that specifies a recommended choice for the user. When the list of choices is displayed, if that index is valid, then some asterisks will be displayed to highlight the recommended choice. Also, at the prompt, the default value will be take in the case that the user simply hits return.

⟨*def get choice index*⟩≡

```
def get_choice_index(choices,default=-1):
   # If the default is out of range, raise an exception
   if default != -1 and (default < 0 or default > len(choices)-1):
      raise CmdError,(-1,("Internal error, default value (%d) was out of range 0..%d"
                          %(default,len(choices)-1)))

   while 1:
      ⟨display choices to user⟩
      sys.stdout.write("Which choice would you prefer? ")
      if default != -1:
         sys.stdout.write(" (hit ENTER for default) ")
      the_choice=sys.stdin.readline()
      if default!=-1:
         # if we match on white space, then we just picked up a default selection
         m=re.match("^\s*$",the_choice)
         if m:
             # we got a blank line, so return the default choice.
             print
             print "you have selected ",choices[default]
             return default
      try:
         choice_num=string.atoi(the_choice)
      except ValueError:
         print
         print "The string you entered: "
         print "  ",the_choice
         print
         print "is not a valid integer. Please try again."
         print
      else:
         if choice_num > 0 and choice_num <= len(choices): break
         else:
            print ("%d is outside the range 1..%d. Please try again" %
                   (choice_num,len(choices)))
            print
   print
   print "you have selected ",choices[choice_num-1]
   return choice_num-1
```

We want to print these choices taking the `default` into account. If the `default` is -1, then we just print out with no asterisks. But if the `default` is one of the index values, then we print "default" in front of the choice.

The key idea here is that `default` is an index value based at 0, not a choice value based at 1.

⟨*display choices to user*⟩≡

```
i=0;
while (i!=len(choices)):
   if i==default:
      print " Default: %4s. %s ***" % (i+1,choices[i])
   else:
      print " %13s. %s" % (i+1,choices[i])
   i=i+1
print
```

1 Synchronizing between ClearCase and Subversion

From an implementation perspective, there are a variety of small functionalities that you will find this script using at one point or another.

- getting command line arguments
- outputting to a log file
- being able to execute SVN commands, get status, collecting output.
- being able to execute ClearCase commands, get status, and collect output.
- walk a tree, collect directories, sort files.
- compare text files and detect if there's a difference or not.
- compare binary files and detect if there's a difference or not.
- tree walk and ignore files matching certain names (.svn).
- determine the net set of moves and changes

⟨*sync*⟩≡
```
⟨copyright⟩
import os,sys,tempfile,string,re,bisect,getopt,fnmatch,filecmp,shutil
⟨def print general usage⟩
⟨def print import usage⟩
⟨def print export usage⟩
⟨class cmd object⟩
⟨class Change⟩
⟨class Change list⟩
⟨class Diags⟩
diags=Diags()
cache_o_rev={}  # caches info about moves for other-branch move changes
⟨def get choice index⟩
⟨def sep to slash⟩
⟨def slash to sep⟩
⟨def subtract longest common suffix⟩
⟨def process url⟩
⟨def run cmd⟩
⟨def clean up after⟩
⟨def sort out list⟩
⟨def parse svn log into change list⟩
⟨def list stuff⟩
⟨def get cview checkouts⟩
⟨def get view private⟩
⟨def chop prefix⟩
```

```
⟨def canonical path⟩
⟨def quoted⟩
⟨def tree compare⟩
#
# -------------------------- Main Program Starts Here --------------------------------
#
⟨get cmdline args⟩
⟨setup logfile⟩
⟨print cc and svn versions⟩
if sys.argv[1]=='import':
   ⟨import stuff⟩
else:
   assert sys.argv[1]=='export'
   ⟨export stuff⟩
diags.print_summary()
sys.exit(0)
```

What we want to do in this chunk is open the log file and redirect `stdout` to that file, in the cases that `stdout` should actually go to that file.

The first thing we do is open `log_file_name` for write output.

⟨*setup logfile*⟩≡

```
class File_tee:
   def __init__(self):
      self.stdout=sys.stdout
      self.log_file=open(log_file_name,'w')
      sys.stdout=self

   def __del__(self):
      sys.stdout=self.stdout
      self.log_file.close()

   def write(self, string):
      self.stdout.write(string)
      self.log_file.write(string)

file_tee_instance=File_tee()  # there's is meant to be just a single instance
```

The assumptions about when this starts are that:

- There are two ClearCase views, marking a before and an after state on the source branch.
- The before state exactly matches the current contents of the Sub Version workspace.

The steps to exporting from ClearCase into a SubVersion workspace are as follows:

1. Compare the ClearCase reference view with the SubVersion workspace. They should be identical.
2. Compare the two ClearCase views in order to determine the delta that must be applied to SubVersion.
3. Resolve any element MOVES from the directory changes detected. Use ClearCase element oids (object identifiers) to do this.
4. Perform the directory change type updates in the SVN workspace.
5. Perform the file change updates.

⟨*export stuff*⟩≡

```
dirs_to_add=[]
dirs_to_del=[]
files_to_add=[]
files_to_del=[]
differences=[]
elements_to_move=[]
tree_compare(ccref_view_root, svn_client_root,
             dirs_to_add, dirs_to_del, files_to_add, files_to_del, differences,1)
⟨notify user of svn and reference view differences⟩
# last tree_compare should have been empty (unless in -force mode)
# now compare the CC view and the svn workspace.

tree_compare(ccview_root, ccref_view_root,
             dirs_to_add, dirs_to_del, files_to_add, files_to_del, differences,1)
diags.start_info(0)
print
print "dirs to add",dirs_to_add
print "dirs to del",dirs_to_del
print "files to add",files_to_add
print "files to del",files_to_del
print "elements to move: ",elements_to_move
print "modified files: ",differences

diags.end()
```

⟨*use oids to resolve moves*⟩
```
# This split up for debugging purposes...
diags.start_info(0)
print
print "dirs to add",dirs_to_add
print "dirs to del",dirs_to_del
print "files to add",files_to_add
print "files to del",files_to_del
print "elements to move: ",elements_to_move
print "modified files: ",differences
diags.end()
```
⟨*apply changes to svn workspace*⟩

In this chunk we process each list from `dirs_to_add` to `differences` in order to perform the changes necessary to actually synchronize the SVN workspace with the current view from ClearCase.

⟨*apply changes to svn workspace*⟩≡
```
for new_dir in dirs_to_add:
    ⟨add new dir⟩
for old_dir in dirs_to_del:
    ⟨del old dir⟩
for new_file in files_to_add:
    ⟨add new file⟩
for old_file in files_to_del:
    ⟨del old file⟩
for source_target in elements_to_move:
    ⟨move elements⟩
for file in differences:
    ⟨update file into SVN workspace⟩
```

In this case, `new_dir` is the universal path to a new directory to be added. This means we need do several things:

1. Form the source and target paths.
2. assert that the source exists and destination does not (but tree-walk already asserts this is true. This isn't much of a test.)
3. Perform a directory copy.
4. change directory into the svn client space
5. issue an add command

⟨*add new dir*⟩≡
```
# need to chop any leading \ char off the front - this is non-OS portable. So what?
if new_dir[0:1]==os.sep:
   new_dir=new_dir[1:]
if new_dir!='':
   destination=os.path.join(svn_client_root,new_dir)
   source=os.path.join(ccview_root,new_dir)
   print
   print ("ADD (for folder) of source (%s)\n\tand dest (%s)" %
              (source,destination))
   if not flag_no:
      shutil.copytree(source,destination)
      os.chdir(destination)
      run_cmd("attrib -R /s /d",
         "Error resetting read-only permissions at\n\t%s" % (destination))
      os.chdir(svn_client_root)
      run_cmd("svn add %s" % (new_dir),
         "Could not do folder add of %s " % (new_dir))

else:
   raise CmdError,(-1,"Unexpectedly, the root itself needs to be created!")
```

This is pretty simple:

1. change directory into the svn client space
2. issue a del command

⟨*del old dir*⟩≡
```
print ("DELETE subtree %s" % (old_dir))
if not flag_no:
   os.chdir(svn_client_root)
   run_cmd("svn del %s" % (old_dir),"Could not do svn del %s " % (old_dir))
```

Like adding `new_dir` but we don't need to `treecopy`, just `copy`.

⟨*add new file*⟩≡
```
# need to chop any leading \ char off the front - this is non-OS portable. So what?
if new_file[0:1]==os.sep:
   new_file=new_file[1:]
assert new_file!=''
destination=os.path.join(svn_client_root,new_file)
source=os.path.join(ccview_root,new_file)
print
print ("ADD (for file) of source (%s)\n\tand dest (%s)" %
           (source,destination))
if not flag_no:
   shutil.copy(source,destination)
   run_cmd("attrib -R %s" % (quoted(destination)),
              "could not turn off read-only attribute on %s" % (quoted(destination)))
   os.chdir(svn_client_root)
   run_cmd("svn add %s" % (quoted(new_file)),
      "Could not do file add of %s " % (quoted(new_file)))
```

This is pretty easy. Just issue the `svn del` command.

⟨*del old file*⟩≡
```
print ("DELETE file %s" % (old_file))
if not flag_no:
   os.chdir(svn_client_root)
   run_cmd("svn del %s" % (quoted(old_file)),
      "Could not svn del %s " % (quoted(old_file)))
```

Just do an `svn move`.

⟨*move elements*⟩≡
```
print ("MOVE of source (%s)\n\tand dest (%s)" % source_target)
if not flag_no:
   os.chdir(svn_client_root)
   run_cmd("svn move \"%s\" \"%s\"" % source_target,
      "Could not svn move %s to %s " % source_target)
```

Copy the changed file into the SVN workspace. That's all! Files are already checked out.

⟨*update file into SVN workspace*⟩≡

```
# need to chop any leading \ char off the front - this is non-OS portable. So what?
if file[0:1]==os.sep:
   file=file[1:]
assert file!=''
destination=os.path.join(svn_client_root,file)
source=os.path.join(ccview_root,file)
print
print ("UPDATE of source (%s)\n\tand dest (%s)" %
           (source,destination))
if not flag_no:
   shutil.copy(source,destination)
```

⟨*notify user of svn and reference view differences*⟩≡

```
# test for any differences, and if there are, abort, unless force mode is on
if dirs_to_add or dirs_to_del or files_to_add or files_to_del or differences:
   if force_mode:
      diags.start_warning()
   else:
      diags.start_error()
   print "The ClearCase reference view ",ccref_view_root
   print "and the svn client workspace ", svn_client_root
   print "do not EXACTLY MATCH:"
   print
   if dirs_to_add:
      print "There are directories missing from the SVN client:\n",dirs_to_add
   if dirs_to_del:
      print "There are extra directories in the SVN client:\n",dirs_to_del
   if files_to_add:
      print "There are files missing from the SVN client:\n",files_to_add
   if files_to_del:
      print "There are extra files in the SVN client:\n",files_to_del
   if differences:
      print "There are files that are not identical:\n",differences
   print
   if force_mode:
      print "Since the -f / --force- option is in effect, changes shall"
      print "be applied in spite of this descrepancy. The results "
      print "ARE UNPREDICTABLE. (consider this fair warning)"
      print
   else:
      print "It is recommended you rectify this situation before attempting"
      print "to export from ClearCase to SubVersion. It is possible (but not"
      print "recommended) to use the -f / --force option to proceed anyway."
      diags.print_summary()
      sys.exit(4)
```

Now we:

- Do a `cleartool dump` on each deleted element to get its oid.
- Do a `cleartool dump` on each added element to see if its oid matches any of the deletes.
- If we get a match, then:
  - assert that this oid hasn't been matched before.
  - delete the add and delete from our lists.
  - add to our move list.
  - record that this match has been made.

⟨*use oids to resolve moves*⟩≡

```
del_oids={}
matched_oids={}   # used for sanity checking
for d in dirs_to_del+files_to_del:
   target_path=os.path.join(ccref_view_root,d)  # this line not same as below
   diags.start_info(1)
   print "get oid of %s" % target_path
   diags.end()
   m=None
   for oid_line in string.split(run_cmd('cleartool dump %s@@' % quoted(target_path),
        'couldn\'t do a dump on %s@@' % quoted(target_path),1),'\n'):
      m=re.match("^oid=",oid_line)
      if m:
         diags.start_info(1)
         print oid_line
         diags.end()
         del_oids[oid_line]=d
         break
   assert m  # we should have found a line saying ^oid=blah blah$
for d in dirs_to_add+files_to_add:
   target_path=os.path.join(ccview_root,d)   # notice the difference at this line
   diags.start_info(1)
   print "get oid of %s" % target_path
   diags.end()
   m=None
   for oid_line in string.split(run_cmd('cleartool dump %s@@' % quoted(target_path),
        'couldn\'t do a dump on %s@@' % quoted(target_path),1),'\n'):
      m=re.match("^oid=",oid_line)
      if m:
         diags.start_info(1)
         print oid_line
         diags.end()
         ⟨handle add oid⟩
```

To repeat what is written above:
If we get a match, then:

- assert that this oid hasn't been matched before.
- delete the add and delete from our lists.
- add to our move list.
- record that this match has been made.

⟨*handle add oid*⟩≡
```
diags.start_info(1)
print "Checking if the add for %s is really a move:" % (d)
diags.end()
if del_oids.has_key(oid_line):
   print "\nAha! Found a MOVE!\n"
   assert not matched_oids.has_key(oid_line)
   matched_oids[oid_line]=1  # just add some dummy value
   if d in files_to_add:
      del files_to_add[files_to_add.index(d)]
   else:
      assert d in dirs_to_add  # this may kill performance!
      del dirs_to_add[dirs_to_add.index(d)]
   # could do a little refactoring here! :-)
   if del_oids[oid_line] in files_to_del:
      del files_to_del[files_to_del.index(del_oids[oid_line])]
   else:
      assert del_oids[oid_line] in dirs_to_del  # this may kill performance!
      del dirs_to_del[dirs_to_del.index(del_oids[oid_line])]
   elements_to_move.append((del_oids[oid_line],d))
```

⟨*import stuff*⟩≡

```
  ⟨get the url from info of the svn workspace⟩
  ⟨declare and process change list from svn⟩
  ⟨scan svn and cc trees and report⟩
  if differences:
     diags.start_info(1)
     print "\nDifferences Are:\n-----------------\n"
     diags.end()
     for i in differences:
       diags.start_info(1)
       print i
       diags.end()
     ⟨update files from differences⟩
  else:
     print "No file changes to update."
```

For each file, we want to check out the file, copy over the original. And that's it. So we have to from the source and destination paths, do a checkout, and then a copy.

⟨*update files from differences*⟩≡

```
for f in differences:
   # need to chop any leading \ char off the front - this is non-OS portable. So what?
   if f[0]==os.sep:
      f=f[1:]
   parent=os.path.join(ccview_root,os.path.split(f)[0])
   source=os.path.join(svn_client_root,f)
   destination=os.path.join(ccview_root,f)
   print
   print "UPDATE of %s from %s" % (source,destination)
   if not flag_no:
      os.chdir(parent)
      # now checkout and then copy
      try:
         cmd=cmd_object("cleartool checkout -c \"Checkout for update by clearsvn.py.\"
         print
         print cmd.out
         print
      except CmdError,(rc,errstring):
         print
         print "stderr=",errstring
         print
         diags.start_error()
         print "Couldn't checkout ",destination
         sys.exit(3)
      else:
         del cmd
      # shutil.copy() can raise an IOError - we do not handle for now.
      shutil.copy(source,destination)
```

The code in this chunk is based on that found for the `getopt` module in The Python Standard Libary book.

This chunk is being revised to support both import and export modes. Note that the tersm "import" and "export" are relative to ClearCase. The tool is called `clearsvn` after all, and not `svn2cc`.

The first thing we do is test `argv` to see if it holds any arguments. If none, we print the general usage help. If at least one, we get that arg. If it is "import" (once `lowered` that is) then we process the rest of the arguments as in import mode. If it is "export", then we do likewise for export mode. If neither of those two are seen as the first arguments, then we print the general usage anyway.

Now, `sys.argv` always has a 0 element that gives the name of this executeable, however it has been invoked. So length is 1 for the case of 0 arguments.

⟨*get cmdline args*⟩≡
```
# initialize the data for command line arguments
svn_range=None
flag_no=0
force_mode=0
svn_client_root=None
ccview_root=None
ccref_view_root=None
svn_tree_path=None
svn_inc_path=None
log_file_name=os.path.join(".","log.txt")
opts=None
args=None
assert len(sys.argv) > 0
if len(sys.argv) == 1:
   print_general_usage()
   sys.exit(3)
if string.lower(sys.argv[1])=='import':
   ⟨get import cmdline args⟩
elif string.lower(sys.argv[1])=='export':
   ⟨get export cmdline args⟩
else:
   print_general_usage()
   if sys.argv[1] not in ('-h','--help'):
      print "ERROR: ",sys.argv[1]," invalid. Expected \'import\' or \'export\'."
   sys.exit(3)
```

The options we process for import are:

- `-h` to print usage or help, which you get on error anyway.
- `-s <dir>` to give the path of the root of the svn client
- `-l <path>` log file path. Defaults to `./log.txt`.
- `-i <path>` This should be the path within the svn repository that is being synchronized. The `-i` path must subsume the `-t` path. They can be the same.
- `-t <path>` to give the path to take off the front of each item from svn that we process. This should be the path within the svn repository the same as the path that was given to svn for establishing the client. For example `/trunk`.
- `-v high,med,low` for verbosity. You turn on `high` and you get a lot.

It would be nice if:

- We had the ability to report all bad bugs by sort of jumping back into the `getopt` routine.
- We had usage printed out in a better way.

⟨*get import cmdline args*⟩≡

```
try:
  opts, args = getopt.getopt(sys.argv[2:], "r:nhs:t:i:c:v:l:",
       ["help","svn-client-root=","svn-tree-path=",
        "svn-inc-path=","ccview-root=","verbosity=","log-file="])
except getopt.GetoptError,mesg:
  print mesg
  sys.exit(1)
# process options

for o,v in opts:
  if o in ("-h","--help"):
     print_import_usage()
     sys.exit(0)
  elif o in ("-s","--svn-client-root"):
     svn_client_root=v
  elif o in ("-t","--svn-tree-path"):
     svn_tree_path=v
  elif o in ("-i","--svn-inc-path"):
     svn_inc_path=v
  elif o in ("-c","--ccview-root"):
     ccview_root=v
  elif o in ("-v","--verbosity"):
```

```
      diags.set_verbosity(v)
   elif o in ("-l","--log-file"):
      log_file_name=v
   elif o == "-n":
      flag_no=1
   elif o== "-r":
      svn_range=v
diags.start_info()
print "svn ROOT ",svn_client_root
print "svn Tree path",svn_tree_path
print "svn Inclusion path",svn_inc_path
print "CC Root ",ccview_root
print
print
diags.end()

# All three of these paths are mandatory on the command line.
# The help/usage obviously needs improving.

if not (svn_client_root and svn_tree_path and svn_inc_path):
   print_import_usage()
if (svn_client_root==None):
   diags.start_error()
   print "no svn client root given (use the -s option)."
if (svn_tree_path==None):
   diags.start_error()
   print "no svn tree path given (use the -t option)."
if (svn_inc_path==None):
   diags.start_error()
   print "no svn inc path given (use the -i option)."
if not (svn_client_root and svn_tree_path and svn_inc_path):
   sys.exit(1)
# we have to use the canonical_path of these paths for comparisons to always be valid
svn_inc_path=canonical_path(svn_inc_path)
svn_tree_path=canonical_path(svn_tree_path)
temp_inc_path=Change(svn_inc_path)
temp_tree_path=Change(svn_tree_path)
if not temp_tree_path.subsumes(temp_inc_path):
   diags.start_error()
   print ("The tree path (%s)\n\tdoes not subsume the inclusion path (%s)." %
                        (svn_tree_path,svn_inc_path))
   sys.exit(1)
```

Has the -l for logfile, just like for import.

⟨*get export cmdline args*⟩≡

```
try:
  opts, args = getopt.getopt(sys.argv[2:], "fnhs:c:b:v:l:",
       ["help","force","svn-client-root=","svn-tree-path=","ccref-view-root=",
        "svn-inc-path=","ccview-root=","verbosity=","log-file="])
except getopt.GetoptError,mesg:
  print mesg
  sys.exit(1)
# process options

for o,v in opts:
  if o in ("-h","--help"):
     print_export_usage()
     sys.exit(0)
  elif o in ("-f","--force"):
     force_mode=1
  elif o in ("-s","--svn-client-root"):
     svn_client_root=v
  elif o in ("-b","--ccref-view-root"):
     ccref_view_root=v
  elif o in ("-c","--ccview-root"):
     ccview_root=v
  elif o in ("-v","--verbosity"):
     diags.set_verbosity(v)
  elif o in ("-l","--log-file"):
     log_file_name=v
  elif o == "-n":
     flag_no=1
diags.start_info()
print "svn ROOT ",svn_client_root
print "CC reference view root",ccref_view_root
print "CC Root ",ccview_root
print
print
diags.end()
# This next chunk intelligently presents error messages if appropriate.
if not (svn_client_root and ccview_root and ccref_view_root):
   print_export_usage()
if (svn_client_root==None):
   diags.start_error()
   print "no svn client root given (use the -s option)."
if (ccview_root==None):
   diags.start_error()
   print "no ClearCase view root path given (use the -c option)."
```

```
if (ccref_view_root==None):
   diags.start_error()
   print "no ClearCase reference view root path given (use the -b option)."
if not (svn_client_root and ccview_root and ccref_view_root):
   sys.exit(1)
```

1.1 Processing Directory Checkouts

Here's the theory of operaton. We need to add several parts to what we have going already:

- Put in an ad hoc kluge to address the bogus Add we have in our logs.
- Filter out stuff that in gathering the log that is out of scope.
- Clip off any prefix on the path of each source item according to how the client space was gotten from the svn repository.
- Keep a list of directories that are known to be checked out.
- As we process each mkelem
  - Get its parent directory. (make sure this works for the top level!)
  - See if the parent directory is on our checked-out list. If so, skip the next step.
  - Check out the parent directory and add it to our list of checked out directories.
  - Add the element. Test what kind it is on the source side, in svn.
    - For a directory, do a `cleartool mkdir`, so as to leave the directory checked out.
    - For a file, copy the file over from svn and then do `cleartool mkelem`.

⟨*test of checkout lsco and lsprivate*⟩≡

```
⟨checkout an element⟩
print "\ncheckouts\n------------\n"
for co in get_cview_checkouts("M:/Bill_svn_test/new_guy/clearcase-sync"):
   print co
print "\nprivate files\n------------\n"
for p in get_view_private("M:/Bill_svn_test/new_guy/clearcase-sync"):
   print p
```

This chunk as currently written changes directory to where we want to do a checkout and then performs a checkout. Some code factoring is probably in order here very soon. Can probably add a method to the `cmd_object` class to better support ClearCase operations. Either that or derive a class. Something.

⟨*checkout an element*⟩≡
```
try:
   os.chdir("M:/Bill_svn_test/new_guy/clearcase-sync")
   cmd=cmd_object("cleartool checkout -c \"Checkout by clearsvn.py.\" .")
   print
   print cmd.out
   print
except CmdError,(rc,errstring):
   print
   print "stderr=",errstring
   print
   diags.start_error()
   print "Couldn't checkout element"
   sys.exit(3)
else:
   del cmd
```

Now we want to find all checkouts in the current view, as well all of the private files.

This may raise `OSError` if the `chdir` fails, or a `CmdError` if the `lsco` fails, but the caller may decide best how to handle such exceptions.

⟨*def get cview checkouts*⟩≡
```
def get_cview_checkouts(view_path):
   cmd=None
   os.chdir(view_path)
   cmd=cmd_object("cleartool lsco -short -cview -avobs")
   return string.split(cmd.out,'\n')
```

⟨*def get view private*⟩≡
```
def get_view_private(view_path):
   cmd=None
   os.chdir(view_path)
   cmd=cmd_object("cleartool lsprivate -short")
   return string.split(cmd.out,'\n')
```

Now we need a couple of helper functions. The first one takes two strings and returns a transformation of the second string. To wit, the first string is expected to be a substring (possibly entirely matching) of the second string. An assertion failure occurs if such is not the case. The second string with the matching first string truncated off the front becomes the return value of this function.

⟨*def chop prefix*⟩≡
```
def chop_prefix(prefix,str):
   ### print "\n\nprefix (%s) and str (%s)\n" % (prefix,str)
   assert prefix==str[:len(prefix)]
   return str[len(prefix):]
```

⟨*def quoted*⟩≡
```
def quoted(strn):
   return "\""+strn+"\""
```

The purpose of this function is simple: return a totally normalized path. We do two things: (1) normalize case to lower case, and (2) use the `os.path normalize` function to flip path separators around and so forth. This guarantees we can compare and sort paths with confidence.

Actually, we are experimenting with leaving the case as is. The theory is that we only get file and path names from sources, like directory listings, and the tools themselves, such that case in file names and paths is accurately reported. That, and the customer wants case preserved.

⟨*def canonical path*⟩≡
```
def canonical_path(aPath):
   result=os.path.normpath(aPath)
   if result=='.':
      result=''  # do this in case the Python rev we're at normalizes to .
   return result
```

This chunk helps validate that we actually have ClearCase and SubVersion in the local environment, as well as provide in the log that the user can see the versions of these tools being used by `clearsvn.py`.

Looks like we have some subroutines what we could factor out of here.

⟨*print cc and svn versions*⟩≡
```
run_cmd("cleartool -version",
  "You don't have ClearCase's cleartool in the path. Is ClearCase installed?")
run_cmd("svn --version",
  "You don't have svn in your path. Is SubVersion installed?")
```

⟨*def run cmd*⟩≡
```
def run_cmd(theCommand,theErrorMessage,info_level=0):
   try:
      cmd=cmd_object(theCommand)
      diags.start_info(info_level)
      print cmd.out
      print
      diags.end()
   except CmdError,(rc,errstring):
      diags.start_error()
      print "stderr=",errstring
      print
      print theErrorMessage
      sys.exit(3)
   else:
      result=cmd.out
      del cmd
      return result
```

⟨*declare and process change list from svn*⟩≡
```
change_list2=parse_svn_log_into_change_list(svn_inc_path,svn_tree_path,svn_range)
print change_list2
change_list2.handle_cc_dir_changes()
```

What we want to do here is invoke the parser, loading up `change_list`.

What we really want to do is make this into a subroutine of sorts. We are going to need to pass in:

- the revision range. This sometimes will come from
-

⟨*def parse svn log into change list*⟩≡

```
def parse_svn_log_into_change_list(the_include_path,the_tree_path,
                                   range_parameter,url=None):
  change_list=Change_list()
  try:
    ⟨get and process svn log⟩
    return change_list
  except ChangeError,mesg:
    print "\nProbably an internal error with the Change class:\n\n",mesg
    sys.exit(2)
  except OSError, mesg:
    diags.start_error()
    print "Your SVN root argument must be bad:\n\n",mesg
    sys.exit(1)
  except CmdError,(rc,errstring):
     diags.start_error()
     print "got return code of ",rc
     print "stderr=",errstring
```

1.2 dealing with directory trees

We are now working on a function which will replace the existing chunks of code so that two trees can be walked and compared via a function:

In subsumption mode, which is off by default, we only add topmost items to our lists. That is, if `a/b` and `a/b/file` need to be added, we only add `a/b`. This gets a little tricky considering there are files and directory lists kept separately. But we handle it all right.

a bug we're working on The `list_stuff` routine is not including the root directory, or `.` in its output. This is as expected. Then, when `sort_out_lists` runs, it does not enter the root directory in `to_scan`. Thus, the root directory is never scanned for source file changes. This should be affecting both import and export.

We might fix this by adding a blank or empty string path to `to_scan`. Or we might add `.` or maybe `\`. We must decide which is the right choice. We must also determine what effect this change will have on each part of the code where `tree_compare` is called.

We would expect to put the new entry on the front of the list, expecting that to be in the proper place according to lexicographic sort order.

the solution The solution involves a few small changes. One change is to initialize the `to_scan` list to the empty string. This specifies the root directory. However, that change alone produces assertion failures in the current revision of this code. We also have to update the `visit` routine in a sort of kluge-looking way so that the `arg.root` variable only gets an `os.sep` char appended to it if it really needs it.

⟨*def tree compare*⟩≡

```
def tree_compare(src_root, dst_root, dirs_add, dirs_del,
                 files_to_add, files_to_del, files_diff, subsumption_mode=0):
    files_to_add[:]=[]
    files_to_del[:]=[]
    ⟨perform tree compare⟩
    files_diff[:]=[]
    ⟨do directories in to scan⟩
```

Really all this chunk is about is:

- invoke `list_stuff` on each tree to get its directories.
- then invoking `sort_out_lists` to perform the actual comparison.

The problem is, we cannot just shove `.` into the `to_scan` list, as we get an assertion error later on when we the treewalk routine looks at the `arg.root`.

⟨*perform tree compare*⟩≡
```
dst_folders=list_stuff(dst_root,1,1,'*.svn')
src_folders=list_stuff(src_root,1,1,'*.svn')
dirs_add[:]=[]
dirs_del[:]=[]
to_scan=['']  # initialize to include the root directory
sort_out_lists(src_folders,dst_folders,dirs_add,dirs_del,to_scan,subsumption_mode)
```

⟨*do directories in to scan*⟩≡
```
for d in to_scan:
   # print "\n\t---> scanning ",d  # debug
   src_dir=os.path.join(src_root,d)
   dst_dir=os.path.join(dst_root,d)
   src_files=list_stuff(src_dir,0,0)
   dst_files=list_stuff(dst_dir,0,0)
   # diags.start_info(1)
   # print "\nsrc files %s from %s" % (src_files,src_dir)
   # print "\ndest files %s from %s" % (dst_files,dst_dir)
   # diags.end()
   files_to_scan=[]
   local_files_to_add=[]
   local_files_to_del=[]
   sort_out_lists(src_files,dst_files,
       local_files_to_add,local_files_to_del,files_to_scan)
   k=0
   while k!=len(local_files_to_add):
      local_files_to_add[k]=os.path.join(d,local_files_to_add[k])
      k=k+1
   k=0
   while k!=len(local_files_to_del):
      local_files_to_del[k]=os.path.join(d,local_files_to_del[k])
      k=k+1
   for wacky in local_files_to_add:
      files_to_add.append(wacky)
   for wacky in local_files_to_del:
      files_to_del.append(wacky)
   for i in files_to_scan:
     ⟨compare file i on both sides storing universally⟩
```

What this chunk does is:

- Forms the full source and full destination path names for the given file.
- Does a comparison of the two files to see if they are different or not.
- If they are not different then we're done.
- Otherwise, we form the "universal" name of the file. That just means the path-relative form of the file name.
- We store that in the `differences` list.

To do all this, we take advantage of the values of `src_dir`, `dst_dir`, and `d`. If it weren't for `d`, we would have to use our idiom for subtracting a prefix off the front of a path.

⟨*compare file i on both sides storing universally*⟩≡

```
src_fullname=os.path.join(src_dir,i)
dst_fullname=os.path.join(dst_dir,i)
# used to print INFO message about comparison here
if not filecmp.cmp(src_fullname,dst_fullname):
   diags.start_info(1)
   print "Files DIFFER.\n"
   print "compared: \n\t%s and \n\t%s" % (src_fullname,dst_fullname)
   diags.end()
   files_diff.append(os.path.join(d,i))
# else: be annoying and report that files are identical
```

⟨*scan svn and cc trees and report*⟩≡
```
dirs_to_add=[]
dirs_to_del=[]
files_to_add=[]
files_to_del=[]
differences=[]
tree_compare(svn_client_root, ccview_root,
             dirs_to_add, dirs_to_del, files_to_add, files_to_del, differences)
if dirs_to_add or dirs_to_del:
   print
   print "The Tree Comparison Yields: \n----------------------\n"
   if dirs_to_add:
      diags.start_warning()
      print "there are directories to be added as follows\n"
      for i in dirs_to_add: print i
   if dirs_to_del:
      diags.start_warning()
      print "there are directories to be removed as follows\n"
      for i in dirs_to_del: print i
if files_to_add or files_to_del:
   print
   print "\nThe Directory Comparison Yields: \n---------------------------------------
   if files_to_add:
      diags.start_warning()
      print "there are files to be added as follows\n"
      for i in files_to_add: print i

   if files_to_del:
      diags.start_warning()
      print "there are files to be removed as follows\n"
      for i in files_to_del: print i
```

⟨*test of list stuff*⟩≡
```
print "testing listing of stuff"
thefiles=list_stuff('d:\\temp\\cl3',1,1,'*base*')
print "the files are:\n"
for f in thefiles:
  print os.path.normpath(f)
```

Taken and adapted from recipe 4.18 from the Python Cookbook.

We may want to adapt the pattern thing to make it an exclude, rather than include type of operation. That may be better suited to our purposes.

This function only returns results with their relative paths, not full paths.

The patterns refer to items that should be excluded, not included.

⟨*def list stuff*⟩≡

```
def list_stuff(root, recurse=1,folder_mode=0,patterns=''):

    if len(root) > 0 and root[-1] != os.sep:
       root=root+os.sep

    class Bunch:
       def __init__(self, **kwds): self.__dict__.update(kwds)

    arg=Bunch(recurse=recurse,pattern_list=patterns.split(';'),
       folder_mode=folder_mode, results=[],root=root)

    def visit(arg, dirname, files):
       ⟨remove matching files⟩
       for name in files:
          fullname=canonical_path(os.path.join(dirname,name))
          if (arg.folder_mode and os.path.isdir(fullname)) or (
              not arg.folder_mode and os.path.isfile(fullname)):
                 # only report the relative path
                 arg.results.append(chop_prefix(arg.root,fullname))

       # Cease recursion if it was not requested
       if not arg.recurse: files[:]=[]

    os.path.walk(root, visit, arg)
    return arg.results
```

⟨*remove matching files*⟩≡

```
index=len(files)-1
while index != -1:
  for pattern in arg.pattern_list:
     if fnmatch.fnmatch(files[index],pattern):
         print "INFO: omitting file or directory: ",files[index]
         del(files[index])
         break
  index=index-1
```

⟨*test of sort out list*⟩≡

```
src=["a","b","cde","cat"]
dest=["cat","b","egg"]
to_add=[]
to_del=[]
to_scan=[]
#print what we got:
print "BEFORE\n------\n"
print "src = \t",src
print "dest = \t",dest
print "to_add = \t",to_add
print "to_del = \t",to_del
print "to_scan = \t",to_scan
sort_out_lists(src,dest,to_add,to_del,to_scan)
#print what we got:
print
print "AFTER\n------\n"
print "src = \t",src
print "dest = \t",dest
print "to_add = \t",to_add
print "to_del = \t",to_del
print "to_scan = \t",to_scan
```

This routine takes two input lists. We establish that they sorted in ascending lexicographic order with the empty string at the front of each list. The other three lists are assumed to be empty. This routine then goes through and does the sort, updating two of the output lists.

We also support subsumption mode. See the description in the following chunk of code.

⟨*def sort out list*⟩≡

```
def sort_out_lists(src_list,dest_list,to_add,to_del,to_scan,subsumption_mode=0):
  src_list.append('')
  dest_list.append('')
  src_list.sort()
  dest_list.sort()
  index_s=len(src_list)-1
  index_d=len(dest_list)-1
  while not (src_list[index_s] == '' and dest_list[index_d] == ''):
     if src_list[index_s] > dest_list[index_d]:
        to_add.append(src_list[index_s])
        index_s=index_s-1
     elif dest_list[index_d] > src_list[index_s]:
        to_del.append(dest_list[index_d])
        index_d=index_d-1
     else:
        assert src_list[index_s] == dest_list[index_d]
        to_scan.append(dest_list[index_d])  # use dest_list arbitrarily
        index_d=index_d-1
        index_s=index_s-1
  if subsumption_mode:
     print "**** subsumption mode  to_add and to_del:",to_add,to_del
     clean_up_after(to_add)
     clean_up_after(to_del)
```

This routine is meant to be called exclusively from `sort_out_lists` for the sole purpose of providing for the subsumption mode. The idea is that when the subsumption flag is set, it means that, for the two lists `to_add` and `to_del`, we don't want there to be any member i that is subsumed by some other member j except for the trivial case where they are in fact the same member.

This routine, which is supposed to receive a list of paths in reverse sorted order, should delete all paths subsumed by some other path (except themselves).

The following describes the steps we need follow. Note that the cases of the list having a length less than 2 are covered quite nicely:

1. Reverse the list, so it is in ascending order.
2. Start an index through the list at 1.
3. while the index is not past the end
   - loop and delete the current item at the index, exiting once the index is past the end, or, the indexed item is not subsumed by its predecessor in the list.
4. advance the index by one

⟨*def clean up after*⟩≡

```
def clean_up_after(sorted_path_list):
   sorted_path_list.reverse()
   index=1
   while index<len(sorted_path_list):
      print "INDEX= ",index
      key_pred=Change(sorted_path_list[index-1])
      while index!=len(sorted_path_list):
         key_current=Change(sorted_path_list[index])
         print "TESTING ",sorted_path_list[index-1]," and ",sorted_path_list[index]
         if key_pred.subsumes(key_current):
            print "DELETING ",sorted_path_list[index]
            del(sorted_path_list[index])
         else:
            break
         #
      index=index+1
```

1.3 processing the output of svn log verbose

We are going to have to find out what directory changes have happened since the last time we synchronized from SubVersion (SVN). To do that, we shall issue a command of the form `svn log --verbose -r n:m`. The `n` and `m` values tell the revision number range we want reported. We then process each line that comes back. We are paying particular attention to lines marked A and D, and especially those A lines including the "from" information.

When we see A, that means add. It could also mean move or copy. When we see D, that means delete, but it could also be part of a move operation. What we do is process each line of the output, from oldest to most recent (i.e., in reverse order of how the output from `svn log` comes back).

Here's how we handle things. First, an incoming operation has up to four members:

- an operation, either A for Add or D for Delete.
- a target path/file.
- Only for Adds, if the add is part of a move, it will have a source path/file that should match the target of an earlier Delete.
- Also only for Adds, there will be a revision id for the source path.

The problem becomes, how do we link Adds from sources to past deletes? We had considered a rather complex scheme whose details won't be entered into further in this document. The simplest answer is to keep track of files in such a way that they are not considered to have intrinsic names. That is, we identify files ultimately by a unique id, and suppose that a file's name depends only on it currently being present in some version of a directory.

Next, we describe the data we store for each target of an Add or Delete. Then we'll say how we utilize that data. It should all make sense soon.

Actually, the id should come from the revision. We're not using the id, in fact. But if we simply parsed the most recent revision number, we could do that as well.

- Original operation: Add or Delete represented by A or D.
- Original target location: given as path/file.
- Most recent location: given as path/file. For an add, this is the current location. For a delete, this is where the file most recently was located.
- Flag telling whether or not the file is currently parented.
- Most recent revision id. Set upon a delete and used when subsequent Adds need to refer to a particular file.

Here are the rules for processing the chronologically ordered Add and Delete operations.

When an Add without a source appears, just create a new file record. Mark it as follows:

- Original operation is A.
- Original target location is target, same for current location.
- It is currently parented.
- Give it a new unique id.

Generate an error message if there is currently a parented file with the same location.

If an Add has a source – and so also a revision ID – then look up all records subsumed by the source location and having the same revision id, and that are currently not parented. If no such records are found, report an error and then treat the Add as the case above where no source is given.

If any matching records are found, then do almost the same thing as for a plain add, except:

1. Mark the record as currently parented.
2. update the record's current location to be the target of the add.

If we get a delete then we search for all existing records subsumed (coming at or below in the tree) by the target of the delete:

- If there are no existing records, add a new record:
  - Set it's revID to the most recently parsed revID.
  - Original operation is a D, for Delete.
  - Original target location is the target of the Delete.
  - Currently not parented.
  - Most recent location is the target of the Delete.
- If one or more records are in fact found, then, for each record:
  - Set it's revID to the most recently parsed revID.
  - Mark it as currently not parented.
  - Note that we do not update the most recent current location.

On a Delete, the most recent current location often will be subsumed by the target of the delete. We leave that info alone, and when we process an Add with a source location, we also iterate over all subsumed records.

We can use a single number for both telling if a record is currently parented as well as its most recent revision id: use -1 for the revision id when the record is for a parented file. The revision id only comes into play for orphaned files.

An invariant of our data structure is that files are unique based on their most recent location, their revId, and status of being parented or not. I don't know whether or not it makes sense to check this after each and every operation or not. We may provide a diagnostic test routine for debugging purposes. Also, at final processing time, we'll certainly be able to tell if the final state of the records has turned out to be accurate or not.

1.4 Processing the file records

After we have processed all input gleaned from the output of the `svn log -v`, we must translate the records into ClearCase commands as follows. There are four cases:

- Record is parented, and original operation was an Add. Add a new file into the current location. The source contents can be found under the SVN tree at the same location.
- Record is parented, and the original operation was a Delete. In which case, schedule a ClearCase move from the original location to the current location. The file will then participate in update through the normal course of tree walking.
- Record is orphaned, and original operation was an Add. Ignore the record! Somebody added the file(s) and eventually changed their mind(s).
- Record is orphaned, and the original operation was a Delete. Schedule a ClearCase rmname of the original element.

2 Coding

We need to be able to create one of these lists, we need to be able to insert into it, maintaining order (which will require a `cmp` operator on the `Change` class, and then we can practice our membership and our scanning capability.

⟨*test a d data structure*⟩≡

```
# do an add change, so, just a target path
c1=Change('PROJ\\bin')
# do a delete, so a path and a revid
c2=Change('prOj/SRC/p2.h',4)
print c1
print c2
if c1.subsumes(c2):
  print "c1 subsumes c2"
elif c2.subsumes(c1):
  print "c2 subsumes c1"
else:
  print "c1 and c2 are unrelated"
```

```
change_list=Change_list()  # Python is a case-sensitive language.
change_list.process('A','src/foo.h')
change_list.process('D','inc/foo.h',7)
change_list.process('A','src/foo2.h',7,'inc/foo.h')
change_list.process('D','src/foo.h',8)
print change_list
change_list.get_cc_moves()
```

⟨*junk*⟩≡

```
change_list.insert(Change('A','src/c/foo.c'))
change_list.insert(Change('A','proj/src/p2.h'))
# change_list.insert(Change('D','abc','from')) # should raise an exception
```

We are creating a class `Change_list`. It creates a `self.data` item that is initially just an empty list. It has an insertion operation whereby a change can be added. A lot of smarts will go into that, by the time we're done!

⟨*class Change list*⟩≡

```
class Change_list:
   def __init__(self):
      self.checkouts={}  # set of checked out CC elements in the cc view
      self.data=[]       # list of Changes (as in "class Change" Changes)
```

⟨*class Change list method handle cc dir changes*⟩

⟨*class Change list method process*⟩

⟨*class Change list method get subsumed*⟩

⟨*class Change list method insert*⟩

⟨*class Change list method get deleted*⟩

⟨*class Change list method transform to move*⟩

⟨*class Change list method repr*⟩

This method returns a list of all the `Change` objects contained within `self` that are currently not parented. This is motivated by wanting to provide the user a list of candidate `Delete` changes that might be used to resolve Move operations from `Adds` with sources.

⟨*class Change list method get deleted*⟩≡
```
def get_deleted(self):
   result=[]
   for change in self.data:
      if change.revId!=-1:
         result.append(change)
   return result
```

This method iterates over the `Changes` in our list and does two things. Primarily, each change interpreted according to the rules in section 1.2. Second, we make sure that there are no duplicates floating around.

⟨*class Change list method handle cc dir changes*⟩≡
```
def handle_cc_dir_changes(self):
    for change in self.data:
        if change.revId!=-1:
            # It's an orphaned change; left in a deleted state
            if change.original_op=='D':
                ⟨handle RMNAME⟩
            # else --> ignore the record, somebody added, and later deleted
            else:
                assert change.original_op=='A'
        else:
            # It's a parented, extant file!
            if change.original_op=='D':
                ⟨handle MOVE⟩
            else:
                assert change.original_op=='A'
                ⟨handle MKELEM⟩
```

What we do in this chunk is establish that the parent directory is checked out. We then perform an `rmname` on the destination.

⟨*handle RMNAME*⟩≡
```
marked_elem=change.most_recent_path
if marked_elem!='':
   # need to chop any leading \ char off the front - this is non-OS portable. So what?
   if marked_elem[0]==os.sep:
      marked_elem=marked_elem[1:]
   parent=os.path.join(ccview_root,os.path.split(marked_elem)[0])
   source=os.path.join(ccview_root,marked_elem)
   print
   print "RMNAME of ",source
   diags.rmname()
   if not flag_no:
      os.chdir(os.path.join(ccview_root,parent))
      ⟨checkout parent directory if needed⟩
      ⟨actually do the rmname⟩
else:
   diags.start_error()
   print "you cannot delete the root directory! something is wrong with the input data
   diags.end()
```

The formation of the source and destination need to be revised. Also, we have to make sure two parents are checked out.

⟨*handle MOVE*⟩≡
```
src_elem=change.original_path
dst_elem=change.most_recent_path
# need to chop any leading \ char off the front - this is non-OS portable. So what?
if src_elem[0]==os.sep:
   src_elem=src_elem[1:]
# need to chop any leading \ char off the front - this is non-OS portable. So what?
if dst_elem[0]==os.sep:
   dst_elem=dst_elem[1:]
src_parent=os.path.join(ccview_root,os.path.split(src_elem)[0])
dst_parent=os.path.join(ccview_root,os.path.split(dst_elem)[0])
source=os.path.join(ccview_root,src_elem)
destination=os.path.join(ccview_root,dst_elem)
print
print "MOVE from %s\nto %s\n" % (source,destination)
if not flag_no:
   ⟨checkout parent directories if needed⟩
   ⟨actually do the move⟩
```

We need to adapt the path to the new element so that it becomes '.' if it is empty. That's to accomodate the case of the root itself being new when passing a command argument to the cleartool `mkelem` command.

To actually add the new element we:

1. Check out the parent directory, if it needs checking out.
2. Detect the type of the element (directory vs. file) by testing the source.
3. If directory, then do a `mkdir` in the parent directory of the new element.
4. If file, copy it over, and do `mkelem` in the parent directory.

Possible exceptions that can arise:

- Any of the cleartool commands fail, like `co`, `mkelem`, or `mkdir`.
- If the copy fails.
- If the source element is not found.

What we need to do here is form a path to the parent, the source, and the dest.

- The parent is the cc view root plus the parent path.
- The source is the svn root plus the `new_elem` path
- The destination is the cc view root plus the `new_elem` path.

⟨*handle MKELEM*⟩≡

```
new_elem=change.most_recent_path
# need to chop any leading \ char off the front - this is non-OS portable. So what?
if new_elem[0:1]==os.sep:
   new_elem=new_elem[1:]
if new_elem!='':
   parent=os.path.join(ccview_root,os.path.split(new_elem)[0])
   # print "svn_client_root ",svn_client_root
   # print "ccview_root",ccview_root
   source=os.path.join(svn_client_root,new_elem)
   destination=os.path.join(ccview_root,new_elem)
   print
   print ("MKELEM of %s\n\twith parent (%s)\n\tsource (%s)\n\tand dest (%s)" %
              (new_elem,parent,source,destination))
   if not flag_no:
      os.chdir(os.path.join(ccview_root,parent))
      ⟨checkout parent directory if needed⟩
      ⟨actually do the mkelem⟩
else:
   print "INFO: omitting creation of root directory. It's unnecessary!"
```

What we do here is take the relative path for our new element, get its parent and test if it's in `self.checkouts`. If not, we perform the checkout and add the parent to the list. Otherwise, we need do nothing: the parent is already checked out.

Note that the current working directory has already been set before we got to this chunk of code.

⟨*checkout parent directory if needed*⟩≡

```
# This code needs to be factored into a subroutine.
if not self.checkouts.has_key(parent):
   self.checkouts[parent]=1  # 1 is an arbitrary value
   # now for our cleartool command idiom
   try:
      cmd=cmd_object("cleartool checkout -c \"Checkout by clearsvn.py.\" .")
      diags.start_info(1)
      print cmd.out
      print
      diags.end()
   except CmdError,(rc,errstring):
      diags.start_error()
      print "stderr=",errstring
      print
      print "Couldn't checkout element"
      diags.end()
   else:
      del cmd
```

This chunk copied and hacked from the one above. It's 5am. I want this working soon.

⟨*checkout parent directories if needed*⟩≡

```
# This code needs to be factored into a subroutine.
os.chdir(os.path.join(ccview_root,src_parent))
if not self.checkouts.has_key(src_parent):
   self.checkouts[src_parent]=1  # 1 is an arbitrary value
   # now for our cleartool command idiom
   try:
      cmd=cmd_object("cleartool checkout -c \"Checkout by clearsvn.py.\" .")
      diags.start_info(1)
      print cmd.out
      print
      diags.end()
   except CmdError,(rc,errstring):
      diags.start_error()
      print "stderr=",errstring
      print
      print "Couldn't checkout element"
      diags.end()
   else:
      del cmd
os.chdir(os.path.join(ccview_root,dst_parent))
if not self.checkouts.has_key(dst_parent):
   self.checkouts[dst_parent]=1  # 1 is an arbitrary value
   # now for our cleartool command idiom
   try:
      cmd=cmd_object("cleartool checkout -c \"Checkout by clearsvn.py.\" .")
      diags.start_info(1)
      print cmd.out
      print
      diags.end()
   except CmdError,(rc,errstring):
      diags.start_error()
      print "stderr=",errstring
      print
      print "Couldn't checkout element"
      diags.end()
   else:
      del cmd
```

This is a copy/modify of the code that does the move, but simpler, no?

⟨*actually do the rmname*⟩≡
```
try:
   cmd=cmd_object(("cleartool rmname -c \"Removed during sync from SubVersion.\" %s" %
                   (quoted(source))))
   diags.start_info(1)
   print cmd.out
   print
   diags.end()
except CmdError,(rc,errstring):
   diags.start_error()
   print "stderr=",errstring
   print
   print (("Couldn't do cleartool rmname of %s" %
                   (quoted(source))))
   diags.end()
else:
   del cmd
```

I'm punting on this somewhat: I'm supposing that as long as my current working directory is inside a view, then I can use fully qualified paths, absolute paths that is, to my source and destination. Here's hoping it works so that I don't have to figure out how to compute relative paths. (Could chdir to the root... but what a bother at 5am!)

⟨*actually do the move*⟩≡
```
os.chdir(os.path.join(ccview_root,src_parent))
try:
   cmd=cmd_object(("cleartool mv -c \"Moved during sync from SubVersion.\" %s %s " %
                   (quoted(source),quoted(destination))))
   diags.start_info(1)
   print cmd.out
   print
   diags.end()
except CmdError,(rc,errstring):
   diags.start_error()
   print "stderr=",errstring
   print
   print ("Couldn't do cleartool move of %s to %s" %
                   (quoted(source),quoted(destination)))
   diags.end()
else:
   del cmd
```

This is the part, described twice above, where we test the source element type and either do a `mkdir` or copy the file over and do a `mkelem`.

⟨*actually do the mkelem*⟩≡

```
if os.path.isdir(source):
   try:
      cmd=cmd_object("cleartool mkdir -c \"Created in update from SubVersion.\" "+
                     quoted(os.path.split(new_elem)[1]))
      diags.start_info(1)
      print cmd.out
      print
      diags.end()
   except CmdError,(rc,errstring):
      diags.start_error()
      print "stderr=",errstring
      print
      print "Couldn't make directory: ",destination
      diags.end()
   else:
      del cmd
      self.checkouts[destination]=1  # since making a new directory means it is checke
else:
   shutil.copy(source,destination) # this can raise an IOError - we no handle for now
   try:
      cmd=cmd_object("cleartool mkelem -c \"Created in update from SubVersion.\" "+
                     quoted(os.path.split(new_elem)[1]))
      diags.start_info(1)
      print cmd.out
      print
      diags.end()
   except CmdError,(rc,errstring):
      diags.start_error()
      print "stderr=",errstring
      print
      print "Couldn't make new file element: ",quoted(destination)
      diags.end()
   else:
      del cmd
```

Now for my really cool `Diags` class of which a singleton, `diags` shall be created by and used within this script. Not only will this add lines of code, which is sort of fun in a weird way, but we can effectively support:

- a verbosity flag on the command line.
- the counting of various types of info.
- provision for testing at any point in our run: have we had an error yet? (sometimes it's good to finish what you're doing even as errors accumulate, other times you should stop and say it's not worth going on unless all was ok so far).
- Ability to report a summary of diagnostic counts, especially at the end of a run.

Basically, we:

- Keep a list of counters of errors, warnings, and info messages.
- Have a verbosity level that tells what to pass through.
- Have a way to shunt `stdout`.
- Provide a begin/end protocol to callers to do the counts and control the shunting.

Verbosity levels are low, medium, and high. At low, just errors and warnings come out. At medium, info messages come out as well. At high, fluffy info messages also come out. As of this writing, the only fluffy messages are the ignored lines from the `svn log`.

What the start and end methods are all about is counting the type of diagnostic message, as well as shunting `stdout` until the `end()` method is invoked. Note that shunting really only occurs for the sake of suppressing INFO and fluffy INFO messages.

⟨*class Diags*⟩≡
```
class FileFaker_DevNull:
   def __init__(self):
      pass

   # This is the 'Pirates That Do Nothing' (from VeggieTales) version of a file
   # implementation.
   def write(self, string):
      # guess I was just kidding when I said I'd process your string. :-)
      pass

class Diags:
```

```
def __init__(self):
   self.error_count=0
   self.warning_count=0
   self.rmname_count=0
   self.verbosity="low"
   self.std_out=None
   self.bit_bucket=FileFaker_DevNull()

def set_verbosity(self,level):
   if not level in ("high","med","low"):
      raise CmdError   # should really be some other kind of exception. Ah well.
   self.verbosity=level

def rmname(self):
   self.rmname_count=self.rmname_count+1

def start_error(self):
   assert self.std_out == None
   self.error_count = self.error_count+1
   sys.stdout.write("\nERROR: ")

def start_warning(self):
   assert self.std_out == None
   self.warning_count = self.warning_count+1
   sys.stdout.write("\nWARNING: ")

def start_info(self,level=0):
   # on low, always block
   # on med, only take level 0 items, the rest are fluff
   # on high, do all
   assert self.std_out == None
   if (self.verbosity == "high" or
     (self.verbosity=="med" and level==0)):
      sys.stdout.write("INFO: ")
   else:
      self.std_out=sys.stdout
      sys.stdout=self.bit_bucket

def end(self):
   if self.std_out:
      sys.stdout=self.std_out
      self.std_out=None  # restored to default condition

def no_errors(self):
   return  self.error_count==0
```

```
def print_summary(self):
   print
   if self.error_count==0 and self.warning_count==0 and self.rmname_count==0:
      print "All OK!"
      print "(no errors or warnings or rmnames)"
   else:
      print "TOTAL ERRORS ..... \t",self.error_count
      print "TOTAL WARNINGS ... \t",self.warning_count
      print "TOTAL RMNAMES .... \t",self.rmname_count
   print
```

This method takes a given path and returns a list of each of the records currently in the `Change_list` that are subsumed by that path, and which are associated with the specified revision level.

What we do is see if the path is found. If it is not found, we're done: empty list. If the path is found, then we loop, until the path is not found or we have walked off the end of the list: we add the current record to the list and advance our index.

⟨*class Change list method get subsumed*⟩≡

```
def get_subsumed(self,path,revision_id):
   # print "checking what is subsumed by ",path
   result=[]
   if len(self.data)>0:
      dummy_change=Change(path,revision_id)  # use this for comparison purposes only
      item_insert_point=bisect.bisect(self.data,dummy_change)-1
      while (item_insert_point != len(self.data)
            and dummy_change.subsumes(self.data[item_insert_point])):
        # print "CHECKING subsumption of ",self.data[item_insert_point]
        if ((self.data[item_insert_point].revId == -1 and  revision_id == -1) or
            (self.data[item_insert_point].revId != -1 and revision_id != -1)):
            result.append(self.data[item_insert_point])
        else:
            print ("\nDEBUG: %s does not match %s" %
                   (self.data[item_insert_point].revId,revision_id))
        item_insert_point=item_insert_point+1
   return result
```

This routine has the knowledge of how to update the sorted list, `self.data`, when a new change, an Add or Delete, comes in. If no `revId` or `source_path` are given, then they are assumed to be -1 (for parented) and `None` respectively.

We should cleanup the given paths as well. We want them normalized. That way, simple string processing can be applied to the paths.

coming up with a factored log parser The way this works now is that when the log is parsed, this script essentially sorts the lines into several different kinds and then takes the appropriate action. Generally, the actions are to accumulate `Adds` and `Deletes` into a couple of lists. Then when a single revision has been seen in the log input, the `deletes` and then the `adds` are processed. The `deletes` are done first so that any `adds` with sources can have the maximum chance of being recognized as moves. In the case of an unrecognized source, we may have a move that was introduced from another branch. This may lead to recursion.

What we need to do then, is make it so that another `Change_list` can be created, and a log parsed, in order to update our cache of revision data, as needed.

⟨*class Change list method process*⟩≡

```
def process(self,op_letter,target_path,revId=-1,
            source_path=None,inc_path=None,tree_path=None):
   assert ((source_path == None and  inc_path == None) or
           (source_path != None and inc_path != None))
   print ("processing %s for target %s " % (op_letter,target_path))
   print " and source of ( ",source_path," )"
   print ("with inc %s and tree %s " % (inc_path,tree_path))
   if target_path != None:
      target_path=canonical_path(target_path)
   if source_path != None:
      source_path=canonical_path(source_path)
   #--- paths now normalized. Provides for ok string processing of paths.
   if op_letter=='A':
      if source_path==None:
          assert revId==-1
          # we have a plain Jane new Add change
          self.insert(Change(target_path))
      else:
          assert revId != -1
          ⟨process Add with a source⟩
   elif op_letter=='D':
      assert source_path==None
      assert revId!=-1
      ### print "DELETE TARGET PATH ",target_path
      to_be_axed=self.get_subsumed(target_path,-1)  # -1 means delete only currently pa
```

```
            # print "\nTO BE AXED :",to_be_axed,"\n\n"

            # This next part figures out if the exact directory (if it is a directory)
            # has been found in to_be_axed, or not. If not, then we better insert
            # a Delete operation to make sure a parent directory is wiped out.
            dummy_record=Change(target_path)  # for use as a search key only
            found = 0
            # Now, update each subsumed record to show it is now deleted.
            for r in to_be_axed:
               if dummy_record == r:
                  found=1
               assert r.revId==-1
               r.revId=revId
            if not found:
               self.insert(Change(target_path,revId))

         else:
            raise ChangeError, "Unhandled op_letter "+op_letter
         # print "D E B U G:::: ",self
```

This section is for processing an add with a source. Our context is the set of parameters we receive when a change comes in.

If the list is not empty, we loop over its contents. For each record:

- determine the index of the record (do this first so we get a match on revision id).
- Set the `revId` to -1 (meaning, currently parented).
- Set `most_recent_path` to the transformed value of the `most_recent_location`. We have to strip off the from whence the thing was deleted, which is given by the source of this Add (it is an Add with a source after all!), and we have to add on the target of the Add.
- Delete the record from `self.data`, using the index.
- Insert the record back into `self.data`.

The problem we've found at the moment is that when an add with a source is processed, we must be careful that for each subsumed element, that we transform its `most_recent_location` by removing the old prefix, and putting on the target prefix.

To process an add with a source, we expect that we can resolve it as a move. If we cannot resolve it as a move, then we treat it as an Add without a source (referred to as a "plain Jane" Add).

Resolving a move means identifying a corresponding Delete that the Add came from. There are two avenues to check in making this identification.

If the source of the Add is within the current include path, then we should expect to find one or more matching Deletes in our current change set.

If the source is falls outside the current include path, we then pursue a (potential) chain of one or more revisions that apply to a branch outside of our include path. There may be multiple revisions we have to examine.

If we fail to find a corresponding move on the the branch, we indicate a warning and treat the Add as a plain Jane add.

Otherwise, we have succeeded in resolving the move and so we update the appropriate Delete operation in our current change set.

⟨*process Add with a source*⟩≡

```
diags.start_info(1)
print "\n======= add with a source ----------\n"
diags.end()
key_inc_path=Change(inc_path)
key_source_path=Change(source_path)  # source_path
print "1st step of add resolution: does the 1st path fall under the 2nd path? "
print
print key_source_path
print
print key_inc_path
```

```
print
if not key_inc_path.subsumes(key_source_path):
   print "...foreign source"
   ⟨let user resolve move with foreign source⟩
else:
   print "...local source"
   ⟨try resolving move in current change set⟩
```

What we want to do here is present the user with a choice, and have them select which of the deletes the Add should come from.

In the future, we might have the system make the suggestion for them. We may need to develop more use cases in the process here.

The main trick that we're going to have to work on is extracting the Deletes from the current change list. We might even be able to factor that code out of another part of this program, where Deletes are handled, though that is not an essential change to make.

The list of Deletes is simply those changes which are currently not parented. It doesn't matter what the original operation was. Or does it? Let's take a look at what we do in the other part of our code where we examine Changes in this way.

No, it doesn't matter what the original operation was. We are simply looking for non-parented `Change` records. We can tell that because their `revId` flag will not be -1. That is a weird way to say something is not parented, buthere are historical reasons for that one.

What we want to do is take the list of `Changes` for deleted paths, and presen them to the user, along with the original `Add` with a source that we have. We also add in a choice that says "leave as an `Add`".

If the user chooses to do an `Add` then we process this `Add` with a source as if it has no source. Just an `Add` and we do not resolve a move – per the user's request.

If the user chooses to resolve the `Add`, then we modify the `Delete` that they have designated so that it is now parented, and therefore, a move.

⟨*let user resolve move with foreign source*⟩≡

```
list_of_deletes=self.get_deleted()
list_of_paths=map((lambda record: record.most_recent_path), list_of_deletes)
list_of_paths.append("Just leave as an Add. Don't resolve to a Move.")
print
print "Please choose how to resolve this Add with a source:"
print "   target is ",target_path
print "   source is ",source_path
print
index=get_choice_index(list_of_paths)
⟨act on the choice⟩
```

In this chunk, we do one of two things. If the user's choice returns an index that equals the length of the `list_of_deletes` list then they chose the plain Jane Add option whereby we do not resolve a move. In that case, we insert an Add to honor the user's choice.

If the user made any other choice, then the index refers to the `Change` record that we need to transfrom from a delete into a move. We do that same type of transformation in another location as well. We may want to factor that code out at this point.

⟨*act on the choice*⟩≡

```
if index==len(list_of_deletes):
   # do the plain Jane Add insertion
   diags.start_info()
   print "User select choice of treating the Add with a source as an Add."
   print "Not a move."
   print
   diags.end()
   self.process('A',target_path)
else:
   # transform the indexed delete into a move using the info of the current
   # Add with a source that we are processing.
   self.transform_to_move(list_of_deletes[index],source_path,target_path,revId,1,0)
```

This is the chunk where we are going to enter the loop and walk the chain of revisions in order to attempt to resolve the move.

We need several variables. revId ... this is the next Rev we are going to explore `original_source` this is the one we started going after `current_source` this is the one, based on the rev's we checked so far, that we will look for in the next Rev we explore target... this is the ultimate one we are going for

so,

We make sure we have cached the data for the next rev. We get the changes that the `current_source` subsumes. We expect to find a single move. We see if that move resolves against the ultimate target. If not, we update revId and current source and continue looping.

⟨*try resolving move with foreign source*⟩≡

```
print "going to resolve Add with foreign source."
print "\n The source path is (%s)\n  from revision %d" % (source_path,revId)
original_source=source_path
current_source=source_path
move_resolved="a string"  # just used as an exception
just_an_add="a string"    # ... used as exception to end the while (1) loop.
try:
   while 1:
      ⟨establish cached data for revision revId⟩
      other_change_list=cache_o_rev[repr(revId)]
      print "---"
      print "Change list for revision ",revId,": ",other_change_list
      print "---"
      other_move_candidates=other_change_list.get_subsumed(current_source,-1)
      # uncommenting the next line forces test case
      #   other_move_candidates=other_change_list.get_subsumed('oeu',-1)
      print "source path here (%s) and target path (%s)"%(current_source,target_path)
      (path1,path2)=subtract_longest_common_suffix(current_source,target_path)
      if len(other_move_candidates) != 1:
         diags.start_info(0)
         print "Have not just 1 move candidate: ",len(other_move_candidates)
         diags.end()
      i=0
      while (i!=len(other_move_candidates)):
         record=other_move_candidates[i]
         if record.original_op == 'D':
            # We only look at Deletes because they represent moves.
            # A's are Adds, which are ok, but don't give us any move resolution.
            # So we ignore the A's and stick to the D's.
            ⟨check the move in other change list⟩
            i=i+1
         else:
            del(other_move_candidates[i])
```

```
        # assert: now there are only 'D' original ops in other_move_candidates
        # we need to update rev_id and current_source and keep looping
        if len(other_move_candidates) > 0:
           if len(other_move_candidates) != 1:
              diags.start_warning()
              print "Expected to get one move candidate, but got ",len(other_move_candida
              print "Just using the first one arbitrarily."
              diags.end()
           print "len of other_move_candidates is ",len(other_move_candidates)
           print "0th record is ",other_move_candidates[0]
           print
           revId=other_move_candidates[0].fromRevId
           current_source=other_move_candidates[0].original_path
           print
           print "CHAINING to rev ",revId," and path ",current_source
        else:
           # no Delete/Move candidates, so this must be an Add that really is an Add
           diags.start_info()
           print "Move resolution trail ends... treating Add with source as Add."
           diags.end()
           self.process('A',target_path)
           raise just_an_add

  except move_resolved:
     print "Move Resolved!"

  except just_an_add:
     print "Add is not a move."
```

What we need to do down here is compute the transformation from our `Add` with a foreign source. Actually, we did that prior to this chunk.

Then we apply that transformation to the `original_path` of each of these moves.

We see if we can match that up with a `Delete` in our self's change list. If so, we have resolved a move.

Note: I'm not even worried about multiple matches here. I should say my reason, but I've forgotten it!

If we resolve even one, then we have eliminated the need for adding as a plain Jane `Add`.

⟨*check the move in other change list*⟩≡

```
# ---- use the move(s) to identify moves on the originating side
print
print "---> We need to apply the mapping of %s -> %s" % (path1,path2)
print "to the following record"
print
print record
# This next line applies the mapping.
potential_delete_path=os.path.join(path2,chop_prefix(path1,record.original_path))
print " look for this one: ",potential_delete_path
key_record=Change(potential_delete_path,revId)
index_to_record=bisect.bisect(self.data,key_record)-1
print
print self.data
print
print "index to record",index_to_record
print
print "key_record", key_record
print
if (index_to_record > -1 and index_to_record < len(self.data) and
      self.data[index_to_record] == key_record):
   # then we have found the long sought delete!
   print "Bingo! We have resolved the move to really be a move."
   print
   print self.data[index_to_record]
   # chop off leading os.sep char, if any
   if len(potential_delete_path)>0 and potential_delete_path[0]==os.sep:
      potential_delete_path=potential_delete_path[1:]
   self.process('A',target_path,revId,
                   os.path.join(tree_path,potential_delete_path),
                   inc_path,tree_path)
   raise move_resolved
```

The proper point at which to remove the prefix on `source_path` is once we have decided is to be used for an attempt at move resolution within the current change set. That's because the paths stored in the current change set all have the tree path already removed.

⟨*try resolving move in current change set*⟩≡

```
if tree_path:
   source_path=chop_prefix(tree_path,source_path)
candidate_move_items=self.get_subsumed(source_path,revId)
if len(candidate_move_items)==0:
   # case of a source that should have been found as a move
   diags.start_warning()
   print "Even though we have an add with a source of ",source_path
   print " there are no operations, like a Delete, under that path."
   print " So this Add with a source cannot be resolved to a move."
   print " ** Treating as a plain Add instead of a Move."
   diags.end()
   self.process('A',target_path)
else:
   for record in candidate_move_items:
      self.transform_to_move(record,source_path,target_path,revId)
```

⟨*class Change list method transform to move*⟩≡

```
  def transform_to_move(self,record,source_path,target_path,revId,add_ok=0,strip=1):
     # ---- transform a delete/gone into delete/present with non -1 revId
     print "** transform record into move ",record
     index_to_record=bisect.bisect(self.data,record)-1
     assert self.data[index_to_record] == record
     # sanity check here
     # Even though we *could* potentially get A ops in here, it would
     # would signify a situation where a file add below a directory
     # add had occurred, for the case of a move. That shouldn't happen.
     # If this ends up being a problem, we just sort and ignore the Adds.
     if not add_ok and record.original_op != 'D':
        diags.start_error()
        print ("Internal problem. Saw an original op of %s when D was expected." %
           (record.original_op))
        diags.end()
     if strip==1:
        # This next step in key to transforming a delete into a move
        # Note that fixed_path means it's repaired
        fixed_path=chop_prefix(source_path,record.most_recent_path)
     else:
        fixed_path=""
     # chop off leading os.sep char, if any
     if len(fixed_path)>0 and fixed_path[0]==os.sep:
        fixed_path=fixed_path[1:]
     if (len(fixed_path)>0):
        record.most_recent_path=os.path.join(target_path,fixed_path)
     else:
        record.most_recent_path=target_path
     record.revId=-1
     record.fromRevId=revId   # may need this for move resolution
     # new record may have new location, so we delete and re-insert
     del(self.data[index_to_record])
     self.insert(record)
     # print "new record ",record
     # print "self ",self
```

- We double check that the `from` path occurs outside our -i include path.
- If so, we have a candidate move from another branch.
- In that case what we want to do here is get the revision number associated with the `from` path.
- We use this to check our dictionary to see if we have the move data for that other revision.
- If we don't have that info, then we invoke the proper `log -v` command to get the interpretation of that log into our dictionary.
- From there, we check to see if the `from` path for our add matches any moves that may have occurred on that other branch.
- If we find a match, then we transform the `Add` to a move and treat it that way.
- Otherwise, we flag the `Add` as an error and process it purely as an `Add` of a new file or directory.

When we get an `Add` with a foreign source, we look up any moves from the originating revision.

It is an error if we don't find such a move, but we try to recover by then re-submitting the `Add` as a plain Jane `Add`.

So, when we discover that move, we know that the most-recent-location has to exactly match the source on our unresolved `Add`. We compute the transformation pair (1st, 2nd) on the original `Add` with the foreign source.

We then apply that transformation – but backwards – to the original-location in the move that we found.

If the rule does not apply, that's an error, indicating that we have a weird move or copy going on. In which case, we default to re-submitting the `Add` as a plain Jane.

With the rule applied, we update our unresolved `Add` to have it's source be the result from applying the rule to the source of the move. We resubmit the `Add`. It should resolve as a move, but if it doesn't that means that it was a copy, or some other.

Note that we are relying on the fact that `Adds` with no found source must have sources that are outside of the current include tree. If the source is within the current include tree, we mark that as an error, and treat as a plain Jane.

And we are rather expecting that this type of `Change_list` update will work recursively when resolving moves that occurred across many branches.

Need to add in the check for subsumption beneath the include tree and do a plain Jane if not.

⟨*unused handle add with non matching source*⟩≡

⟨*establish cached data for revision revId*⟩≡
```
if cache_o_rev.has_key(repr(revId)):
   print "We have the move info for revision ",revId
else:
   diags.start_info(1)
   print "We need to load our cache with info for revision ",revId
   diags.end()
   really_good_url=process_url(source_path)
   diags.start_info(1)
   print "--> URL: ",really_good_url
   diags.end()
   cache_change_list=parse_svn_log_into_change_list('','',
         "%d:%d" % (revId,revId),really_good_url)
   cache_o_rev[repr(revId)]=cache_change_list
```

Now what we have to do is some path manipulation. We have to take the URL we got, we have to subtract the trunk as a path. This means removing the longest common suffix. That must result in the tree-path vanishing.

It is an error if it does not.

We must then take the source path we have and suffix it to the URL.

All the while, we have to manipulate this as a directory. So we will introduce a companion routine to `sep_to_slash` that is `slash_to_sep`.

We convert the URL to having `os.sep` separators.

We invoke the subtract-longest-common-suffix routine, passing in the URL, and the tree-path for the root of the svn workspace.

We check that the tree part vanishes.

We don't actually then join the URL with the source path. Turns out that is self-limiting and causes problems: empty log results.

Finally, we convert the result back to having forward slashes.

All in all, three new functions.

⟨*def process url*⟩≡
```
def process_url(source_path):
   url_as_path=slash_to_sep(svn_url)
   (url_as_path,dummy)=subtract_longest_common_suffix(url_as_path,svn_tree_path)
   if dummy != '':
      raise CmdError,(-1,"Must have a bad -t tree-path of %s" % (svn_tree_path))
   return sep_to_slash(url_as_path)
```

⟨*def slash to sep*⟩≡
```
def slash_to_sep(given_path):
   return given_path.translate(string.maketrans('/',os.sep))
```

Given that the paths are in format using the `os.sep` chars, this algortithm can be rather string based. We have an invariant that says we know the index farthest from the left that is where a suffix begins with a slash, or, is entirely empty.

We init by setting these indices to mark the end of the given strings. While neither one hits 0, and while the chars they index in the path strings do match, we decrement them. If the chars they now reference are `os.sep` then we update the mark points.

When we are done, we return each string up to its mark point. Which means they will have no terminating `os.sep` char, unless both paths mismatch starting at the last character, and one of them does end in `os.sep`.

⟨*def subtract longest common suffix*⟩≡

```
def subtract_longest_common_suffix(path_a,path_b):
   left_mark_a=len(path_a)
   left_mark_b=len(path_b)
   index_a=left_mark_a
   index_b=left_mark_b
   while (index_a != 0 and index_b != 0 and
           path_a[index_a-1] == path_b[index_b-1]):
      index_a=index_a-1
      index_b=index_b-1
      # just check one of the strings since they are equal at the index points
      if path_a[index_a] == os.sep:
          left_mark_a=index_a
          left_mark_b=index_b
   return (path_a[0:left_mark_a],path_b[0:left_mark_b])
```

The problem with the `CmdError` exception gets handled farther up the chain, but not in a way that leads to proper reporting. We really should look at the few cases of exception handling in this script and clean it up. Sounds like a deferred change request to me!

⟨*get the url from info of the svn workspace*⟩≡
```
os.chdir(svn_client_root)
cmd=cmd_object("svn info")
info_lines=string.split(cmd.out,'\n')
svn_url=None
for line in info_lines:
   m=re.match("^URL:\s(.+?)\s*$",line)
   if m:
      svn_url=m.group(1)
      break
if svn_url:
   diags.start_info(0)
   print "Using subversion url of (%s)" % (svn_url)
   diags.end()
else:
   # This exception doesn't work that well, but we are still down and dirty. Right?
   raise CmdError,(-1,"Did not find url in svn info output:\n"+cmd.out)
```

Takes a string, presumes it is a path in the current operating system's environment, returns a string in which all the `os.sep` chars have been converted to forward slash characters. What is so useful about this function? It allows paths to be suffixed to a URL which URL needs to have forward slash separators.

⟨*def sep to slash*⟩≡
```
def sep_to_slash(given_path):
   return given_path.translate(string.maketrans(os.sep,'/'))
```

⟨*old handle add with non matching source*⟩≡
```
# this means we got an Add with source, but nothing  matches. This may be
# a move from another branch.
diags.start_error()
print "got Add with source of %s but no items have been deleted from that path" % (sou
if source_path[0]==os.sep:
   print " processing this Add as an Add and not a Move."
   self.process('A',target_path)
else:
   print (" ignoring path (",target_path,") as it lacks a %s as its first char." %
        (os.sep))
```

Just print out each item in order.

⟨*class Change list method repr*⟩≡
```
def __repr__(self):
   result="---- list start ----\n"
   for item in self.data:
      result=result+repr(item)+"\n"
   result=result+"---- end ----\n"
   return result
```

It would be nice to assert somewhere that the type of the second argument is a `Change` object.

This method will take on a lot of intelligence by the time we're through with it. For now, we want to insert into `self` and maintain order, by means of `bisect.insort()`.

We do a sanity check. For incoming items, only those for Add operations should have from, or `source_path` paths, if at all. It may be other types of changes created as a result of doing an `insert` have a from path, but that's ok.

⟨*class Change list method insert*⟩≡
```
def insert(self,new_change):
   # print "Got new change: ",new_change

   item_insert_point=bisect.bisect(self.data,new_change)
   is_present = (
     self.data[item_insert_point-1:item_insert_point] == [new_change])

   if not is_present:
      bisect.insort(self.data,new_change)
   else:
      diags.start_error()
      print "Internal Message: potential extra insert into the change list"
      diags.end()
```

We need a class that can be modeled as a tuple. We intend to use it in a sorted list maintained by `bisect`. We will also hang our `subsumes` comparison operation off of this class as well.

This class needs two constructors, or `__init__` methods. One will be for creating `Change` records based on Add operations. That one will take a single argument as a path. The other will take a path and a revision id and be for setting up a new Delete `Change` record.

⟨*class Change*⟩≡
```
ChangeError="Error occurred in method of class Change"
class Change:
```
⟨*Change init*⟩

⟨*Change method cmp*⟩

⟨*Change method repr*⟩

⟨*Change method subsumes*⟩

⟨*Change method cmp*⟩≡
```
def __cmp__(self, other):
    if self.most_recent_path < other.most_recent_path:
        return -1
    elif self.most_recent_path > other.most_recent_path:
        return 1
    elif ((self.revId == -1 and other.revId == -1) or
          (self.revId != -1 and other.revId != -1)):
        return 0
    elif self.revId == -1:
        return -1
    else:
        return 1
```

Note that we always normalize all paths that we accept into a `Change`. This is for a number of reasons:

- We can sort on the names and store `Change` records in a `Change_list`.
- We can use the `commonprefix` method in determining if one path subsumes another.

⟨*Change init*⟩≡
```
def __init__(self,target_path,revision_id=-1):
   self.revId=revision_id
   if revision_id == -1:
      # being created on an Add
      self.original_op='A'
   else:
      # being created for a delete
      self.original_op='D'
   self.fromRevId=-1 # initial value
   self.original_path=canonical_path(target_path)
   self.most_recent_path=self.original_path
```

The purpose of this method is to return a true value if and only if the `other_change` is holds a `target_path` that is equal to or would fall under the `target_path` of the `self` change object.

The best way to do this comparison is by first splitting both paths into their component parts, in order. Then, while both lists are not empty and the front of each list has equal elements, we keep pulling off the front elements of each list.

If we get both lists are empty, the paths were the same, and so we return true.

If we get that only `self`'s list became empty, then we also return true. Which means if we get that only `other_change`'s list became empty, we return false.

The only other case left is that we got an inequality, and neither list is empty. In this case also we return false.

⟨*Change method subsumes*⟩≡
```
def subsumes(self,other_change):
   # print "checking subsume of ",self,other_change
   self_array=string.split(self.most_recent_path,os.sep)
   other_array=string.split(other_change.most_recent_path,os.sep)
   while (len(self_array) != 0 and len(other_array) != 0 and
           self_array[0]==other_array[0]):
      self_array[:]=self_array[1:]
      other_array[:]=other_array[1:]
   return (len(self_array)==0)
```

We want to print a `Change` in such a way that all its fields have their values shown.

⟨*Change method repr*⟩≡
```
def __repr__(self):
   result=       "    Original Operation: "+self.original_op+"\n"
   result=result+"  Most Recent Location: "+self.most_recent_path+"\n"
   result=result+"         Original_path: "+self.original_path+"\n"
   result=result+"           revision id: "+repr(self.revId)
   if self.revId==-1:
      result=result+" (currently parented) "
   result=result+"  "
   result=result+"from rev id: "+repr(self.fromRevId)
   result=result+"\n"
   return result
```

Now, in order to get the output from `log --verbose`, we most likely need to change directories.

First, let's just try forming a directory using the `os.path` module.

We need to write `d:\\` on the first element of the path because for some reason the `os.path` function does not put the `\\` in there for us.

⟨*form directory changes*⟩≡
```
dir_svn_root=os.path.join('d:\\','svnstuff','cl3','clearcase-sync')
print "SVN root dir = ",dir_svn_root

# how the heck do we know if this fails or not? By an exception being raised.
try:
  os.chdir(dir_svn_root)
  cmd=cmd_object("svn log -v -r HEAD:3")
  ⟨process log in reverse order⟩
except ChangeError,mesg:
  print "\nProbably an internal error with the Change class:\n\n",mesg
  sys.exit(2)
except OSError, mesg:
  diags.start_error()
  print "Your SVN root argument must be bad:\n\n",mesg
  sys.exit(1)
except CmdError,(rc,errstring):
   diags.start_error()
   print "got return code of ",rc
   print "stderr=",errstring
```

This chunk provides a test string that is useful for working on the log parsing routine.

⟨*test string*⟩≡
```
test_string="""------------------------------------------------------------------------
r21 | rjenkinsdomain | 2004-06-13 23:23:11 -0400 (Sun, 13 Jun 2004) | 1 line
Changed paths:
   D /branches/MIPdev4.2/jndi/SlideDirContext.java
   A /branches/MIPdev4.2/jndi/SlideDirtContext.java (from /branches/MIPwork4.2/jndi/Sli
   A /branches/MIPdev4.2/jndi/addwmod2.txt (from /branches/MIPwork4.2/jndi/addwmod2.txt

Add a file, modified/renamed a file for test 6+ from the work branch
------------------------------------------------------------------------
"""
```

⟨*old test string*⟩≡
```
test_string="""------------------------------------------------------------------------
r11 | brassieur | 2004-06-11 10:58:58 -0700 (Fri, 11 Jun 2004) | 2 lines
Changed paths:
   A /trunk/Leo (from /trunk/newBill:9)
   D /trunk/newBill

more test

------------------------------------------------------------------------
r10 | brassieur | 2004-06-11 10:58:36 -0700 (Fri, 11 Jun 2004) | 2 lines
Changed paths:
   D /trunk/Leo
   A /trunk/newLeo (from /trunk/Leo:9)

test

------------------------------------------------------------------------
r9 | brassieur | 2004-06-11 09:48:38 -0700 (Fri, 11 Jun 2004) | 2 lines
Changed paths:
   D /trunk/bill
   A /trunk/newBill (from /trunk/bill:8)

Try first step of a move.

------------------------------------------------------------------------
"""
```

We want to process lines in reverse order and test them to find the ones that have the Add and Delete operations. We can do other operations later to reconcile with what we find when we do our directory walk for file difference comparisons.

We split `cmd.out` on newlines. Then we process that list in reverse order.

And then we want to be able to match the line against a regular expression that can match on the `Add` and `Delete` operations in the log. We suppose that somebody could get cute and enter a comment that would match the same pattern as these operation strings.

The thing to do would be to actually process the output from `cmd.out` of the `svn log --verbose` in order, being careful to capture the changed paths and ignore anything in the comments that are trying to fool us. Push each change line onto a list. Then process the list in reverse order. Quite easily done. But we shall stick with the down and dirty approach for now.

The regular expression is designed with the following in mind:

- There are three spaces, a letter indicating an operation, and another space.
- Then follows a path, which may include embedded spaces (oh joy!).
- Optionally: a (from xxx:n) substring.
- The xxx part gives a path, and n is the revision it comes from. Note the from part only should occur for an A operation.

Note that we have to be careful to use the `canonical_path()` processed form of `m.group(2)` and `m.group(4)` so that our comparisons are valid.

This will be our new chunk for being able to process an `svn log -v` output. We shall begin by creating regular expressions that can recognize:

- A dash line with exactly 80 dashes.
- A change line. If the change is an A or a D, then store it in the list it belongs in.
- An info line. Process the D's, then A's, and then clear the lists. For each A, pass in the number for the rev.
- Or a blank line. Clear the two lists whether they need it or not. This rule helps eliminate bogus actions by people being cute with comments.
- Changed paths. Don't have to do anything.

It should always go change, changed paths, info, dash, blank or comment.

We could define a little local class for a singleton state machine object that knows how to keep up with this type of pattern. This would enhance our ability to detect crap in the input. That's generally helpful to the user. Alternatively, we could invest in using the SVN API, rather than accing SVN via its command line.

What we're going to do is, when we match a line for a change, we shall store the results of the match as a tuple. Often, some of the members of that tuple

will be None. This is to be expected. We will then process the tuples at the appropriate time: either from add_list or from del_list. We can assert the tuple has the expected command letter ('A' or 'D').

⟨*get and process svn log*⟩≡

```
add_list=[]
del_list=[]
# 666 is an arbitrary and highly conspicuous integer greater than 0
universal_revision_number=666
# need this next item for testing for inclusion of changes
key_svn_inc_path=Change(the_include_path)
cmd=None
if url:
   cmd=cmd_object("svn log -v -r %s %s" % (range_parameter,url))
else:
   os.chdir(svn_client_root)
   cmd=cmd_object("svn log -v -r %s" % (range_parameter))
log_lines=string.split(cmd.out,'\n')
# log_lines=string.split(test_string,'\n')
log_lines.reverse()
for line in log_lines:
   ⟨match a change line⟩
   ⟨match a dash line⟩
   ⟨match an info line and process⟩

   m=re.match("^\s*$",line)
   if m:
      ## diags.start_info(1)
      ## print "BLANK"
      ## diags.end()
      if add_list or del_list:
         diags.start_warning()
         for i in add_list:
            print
            print "Discarding CHANGE line due to syntax anomaly in svn log output:"
            print "--> ",i
         for i in del_list:
            print
            print "Discarding CHANGE line due to syntax anomaly in svn log output:"
            print "--> ",i
         diags.end()
      add_list[:]=[]
      del_list[:]=[]
      continue
   m=re.match("^Changed paths:$",line)
   if m:
```

```
      # diags.start_info(1)
      # print "CHANGED PATHS"
      # diags.end()
      continue
   diags.start_info(1)
   print "COMMENT --> ",line
   diags.end()
```

This chunk is where we cal the `process` method of `change_list` to insert the `adds] and [[deletes` into the `change_list` data structure.

⟨*match an info line and process*⟩≡

```
m=re.match("^r(\d+)\s\|\s.+?\|\s\d\d\d\d-.*?\s\|\s\d+\slines?\s*$",line)
if m:
   diags.start_info(1)
   print "INFO LINE: rev is ",m.group(1)
   ### print "processing Adds and Deletes, deletes first"
   diags.end()
   for i in del_list:
     assert i[0]=='D'
     change_list.process('D',i[1],universal_revision_number)
   del_list[:]=[]
   for i in add_list:
      assert i[0]=='A'
      if i[2] == None:
         # the from path for the Add is empty in this case
         change_list.process('A',i[1])
      else:
         # there is a from path for this Add: it's gotta be a MOVE
         change_list.process('A',i[1],string.atoi(i[3]),i[2],
              the_include_path,the_tree_path)
   add_list[:]=[]
   continue
```

⟨*match a change line*⟩≡

```
m=re.match("^   ([A|D|M|R|?]) (.+?)( \(from (.+):(\d+)\))?$",line)
if m:
   diags.start_info(1)
   print " CHANGE ",line
   diags.end()
   #see if we even process this change
   key_target_path=Change(canonical_path(m.group(2)))
   # print "key target_path ",key_target_path
   if key_svn_inc_path.subsumes(key_target_path):
      target_path=chop_prefix(the_tree_path,canonical_path(m.group(2)))
      source_path=None
      if m.group(4):
         temp_cp4=canonical_path(m.group(4))
         # diags.start_info(1)
         # print "the_tree_path is ",the_tree_path
         # print "and canonical form of the source path is ",temp_cp4
         # diags.end()
         source_path=temp_cp4
      if m.group(1)=='A':
         # see the loop variable 'i' below - it's related
         add_list.append((m.group(1),target_path,source_path,m.group(5)))
      elif m.group(1)=='D':
         print "ADDING A DELETE for target ",target_path
         del_list.append((m.group(1),target_path))
      elif m.group(1)=='R':
         if source_path==None:
            diags.start_warning()
            print "DEBUG: ignoring R(eplace) line with no source path"
            print
            diags.end()
         else:
            diags.start_info()
            print "Treating R(eplace) change as a plain Add."
            diags.end()
            add_list.append(('A',target_path,None,None))
      else:
         diags.start_info(1)
         print "ignoring log line: ",line
         diags.end()
   else:
      diags.start_info(1)
      print ("Not in inc path (%s) so \n  ignoring log line: %s " %
               (the_include_path,line))
      diags.end()
   continue
```

⟨*match a dash line*⟩≡
```
m=re.match("^\-{72,72}$",line)
if m:
   diags.start_info(1)
   print "DASH ",line
   diags.end()
   continue
```

Now we need a chunk that demonstrates using `cmd_object`.

⟨*use safe cmd object*⟩≡
```
try:
   cmd=cmd_object("dir /w")
   print "stdout=",cmd.out
except CmdError,(rc,errstring):
   diags.start_error()
   print "got return code of ",rc
   print "stderr=",errstring
```

This code adapted from a posting to ActiveState's ASPN by Tobias Polzin.

I think the real thing to do first is show that we can execute an SVN command to get the history of what has happened. And then from that, put together our set of commands for updating directories in ClearCase.

Due to use of the temp files, this might be a bit slower than other means of executing shell commands from Python, such as what is used in `clearcvs.py`. However, we like getting the return code back.

⟨*class cmd object*⟩≡

```
CmdError='Error running command'

class cmd_object:
    """This is a deadlock safe version of popen2 (no stdin), that returns
    an object with errorlevel,out, and err"""
    def __init__(self,command):
        outfile=tempfile.mktemp()
        errfile=tempfile.mktemp()
        self.result=os.system("( %s ) > %s 2> %s" %
                              (command,outfile,errfile))
        try:
           myfile=open(outfile,"r")
           self.out=myfile.read()
           myfile.close()
           try:
              os.remove(outfile)
           except:
              print "woops"
        except IOError:
           self.out=""
        try:
           myfile=open(errfile,"r")
           self.err=myfile.read()
           myfile.close()
           try:
              os.remove(errfile)
           except:
              print "woops"
        except IOError:
           self.err=""
        if self.result != 0:
           raise CmdError,[self.result,self.err]
```

2.1 Questions

What SVN command(s) do we use to tell to what revision a given client workspace is currently synchronized? We can use `svn info` at the root level, and optionally descend through the entire set of directories and collect a max value.