PERSONAL VERSION CONTROL

File: PersonalVersionControl.odt
Updated: 2025.08.23

This document describes a personal version control suite implemented by Python scripts. It embodies the structure defined by LinkXall, which extends LibreOffice hyperlinking, but may be read by any program that accepts ODT documents, regardless of whether or not it has the LinkXall library. LinkXall is not needed to follow hyperlinks within this document but there are also hyperlinks into non-document files, which only LinkXall can follow.

TOPICS

INTRODUCTION

[Topic End] [Topics]

Most version control programs are designed to support multiple collaborators working on a single program or document and are excessively complicated and bureaucratic for a single developer, especially with a project that involves multiple conceptually related files that that not components of a single program. What's more, to keep track of revisions, they use an internal database whose files are indecipherable, making it difficult to access the repository through any means other than the version control program, leaving users with little means of diagnosing problems or performing an unanticipated operation.

Personal version control is much simpler and more flexible. It uses only standard file system mechanisms and easily understood plain text files and is implemented as Python scripts, exposing its entire operation to scrutiny and project-specific variations to simplify routine use. It is implemented in two flavors. bv.py (backup version) implements a traditional "rolling rev" backup where each version directory contains a snapshot of all project files. This is commonly done manually but the script is easier to use and encourages consistent organization of the repository. Having all files in each directory simplifies using standard tools, like file copy and compare, directly in the repository, with no dependence on the version control system. The main problem with this approach to version control is that it wastes file space on duplicates of files that don't change. dv.py (delta version) addresses this by storing in each version directory files that have changed from previous versions but only a reference to ones that have not changed. The project files that comprise a particular version can always be found by searching the repository but dv.py provides functions to do this automatically. A coincidental advantage of the delta version is that it inherently shows project history. The dv.py -src operation [Src] shows the source of every file in a version, essentially showing how the project has evolved.

Many projects comprise multiple sub-projects, for example code, documentation, support tools, and test programs. These are all related but develop at different rates, making a single version thread inefficient and/or abstruse. Git explicitly supports only a single project with the repository located under the project's development directory. Subversion's central repository for different projects is better suited to a project comprising sub-projects but it is complicated and doesn't clearly distinguish between related sub-projects and unrelated projects.

Except for de-duplication, bv.py and dv.py implement the same repository organization, which, by default, comprises the root directory bak under the project development directory (CWD) and, under bak, version directories named by the user. Similarly to Subversion, semi-independent sub-projects are supported but they are transparently identified by directory name rather than a complex collection of inscrutable directories and files. Each sub-project thread has a unique root (group) name with a suffix identifying the version. For example, the main line of development might be called simply v with a numeric suffix, i.e. v0, v1, etc. The documentation thread might be doc0, doc1, etc; the tools thread tools0, tools1, etc; and the test thread test0, test1, etc. Because this organization is entirely realized in standard directories, it could be implemented manually, but the automation provided by the scripts reduces this effort without reducing the user's freedom to define and manage the repository. dv.py provides additional facilities to manage referential storage, which would be impossible to do manually.

For bv.py, which stores a copy of every file in each version, dividing a complex project into sub-projects (groups) that evolve at different rates can significantly reduce memory consumption. dv.py inherently doesn't waste memory on files that don't change but dividing a project into groups can provide a useful overview of each group's evolution, most significantly revealing the "tip" of the group. If you are not sure of how different potential groups will evolve, using dv.py instead of bv.py can eliminate memory consumption from the group division decision.

bv.py and dv.py are not installed but simply copied to an exe path directory, for example /usr/local/bin in Linux and C:\cmdtools (in PATH) in Windows. In Linux they must be made executable by sudo chmod +x.

To top of [INTRODUCTION]

USE

[Topic End] [Topics]

Personal version control (PVC) is intended to be easier to use than other version control systems but to also support more sophisticated operations, for which you should read this entire document. Basic use requires much less effort. In any case, you should read the Introduction.

Most serious computer users have personal documents divided into business, financial, medical, and contact groups. Technical users can also have technical information groups and programmers project groups. Putting all of these under typical version control could be overwhelming but PVC makes it easy. Although dv.py is more complicated than bv.py, that it doesn't duplicate any file actually makes it easier to use because you can put all of the files in any directory without worrying about wasting storage space.

No preparation is required to put a directory under version control. In a directory containing files you would like to put under version control for the first time, invoke dv.py, which will ask "Do you want to

record all files here in *abspath*/bak/v0?[y/n]". By default, the version directory is bak/v0 under your working directory, for which the message shows the absolute path to be precise about what would happen. Answer Y. dv.py lists all of the files it puts to the version directory, which it creates. For this first version, all files are copied into the directory. The next time dv.py is invoked it will name the version v1. It will "put" all of the files to this but actually copy only ones that have changed or are new. It tells you which ones haven't changed by listing them in [ ] braces. It includes in each version directory the version dictionary, vdict, which tells the version source of each file. If there are no new or changed files (after asking if you want to put the files to v1) it will say "All files are identical to those in v0. Do you want to make v1 anyway?[y/n]". There is rarely any reason for answering Y to this but dv.py always tells you what it would do by default and affords you an opportunity to accept or reject the action.

With many directories under version control it is easy to forget what has changed in a particular one since its most recent put ("tip"). Both bv.py and dv.py provide a version files compare function, e.g. dv.py VERSION -c. If VERSION is blank the working directory and most recent version are compared. Even more often we forget what is different between two specific versions. For this the command is dv.py VERSION -c OTHER.

Version control of program files is often used to find the version in which an error first surfaced and to compare its files to those of the previous version. dv.py and bv.py provide functions that facilitate these investigations. However, they are not useful for documents, which are individually complex and unique. Document version control is usually used to restore some content that has been overwritten. You need the program that creates the documents to do this. With bv.py every version contains all files, simplifying finding any particular version to open. dv.py uses storage space much more efficiently but complicates finding a particular document version. However, every unique version exists in the repository. For any named version you can find the source of its files using the command dv.py VERSION -s. If VERSION is blank, the sources of all files in the most recent version are shown. With this directory information, you can open the document in the appropriate program.

For more complex operations you should read CommandLine. You might want to immediately start taking advantage of these but you might want to first see them work in a test venue before entrusting your files to them. One way to do this would be to make a copy of one of your directories and experiment in it. Testing every feature in your own directory would require considerable effort to translate the description into a specific procedure. The PVC project has already done this for its own regression testing. These tests not only show how the features work but also why they exist and should give you confidence in them. Testing is performed by a variety of Python scripts. You don't have to know Python to study these because Testing fully describes their operation without referencing the code.

The PVC package is a (ZIP) archive of the files that comprise the PVC development directory, which uses dv.py for its own version control. You can unzip this in any directory. In Windows the scripts are automatically executable and can be invoked just by their base name, e.g. "testall". In Linux, each script must be made executable by e.g. chmod +x testall.py (you might want to use 777 instead of +x) and invoked with more path detail, e.g. "./testall.py". The directory includes bvcfg and dvcfg bv and dv project configuration files [Configuration]. These provide an example of how a little preparation can simplify and ensure consistent routine use. Configuration files only define the main version control group, which is usually adequate for monochromatic directories like those containing only documents. Most program projects comprise sub-projects that develop at different rates and are best served by unique group names. put.py in the PVC project illustrates how this sort of version control can be

automated. It puts all py files and dvcfg and bvcfg to the dvc+ group and all odt files to the doc+ group. Using dv.py particularly facilitates using one script to put all sub-project version groups because of its warning about groups that haven't changed and don't need to be put.

The PVC project has only two sub-project version groups but a complex project may have many more and using only one script to put all of them may be inconvenient. The LinkXall LibreOffice library project provides the most extreme example of this. The main sub-project is a Libre-Basic library. The actual library is located in a directory determined by Libre's macro organizer, which is not good for a project directory because it has restricted access and may disappear when Libre is updated. The project directory is more accessible and stable but is not inherently synchronized to the library directory. The project's putlib.py uses dv.py's special AlternateRepository means to put the working library to the lib+ group in its own bak repository. replib.py gets the latest or named lib version to the working library using dv.py's standard Get operation, which allows any directory to be specified as the destination. LinkXall documentation is also developed in the project directory and putdoc.py puts these specific odt files to the doc+ group. Other odt files are used for testing. New test documents are often created to help fix a problem or develop a new feature. To avoid having to repeatedly change the script, puttest.py takes advantage of the Files exclusion clause to put all odt files except the documentation to the test+ group. puttools.py puts all script files (py, bat, sh) and the plain text help file, which describes them all, to the tools+ group. Sometimes the library, documentation, and test document groups are all involved when working on a particular feature. To make sure that changes are captured, put.py invokes putlib.py, putdoc.py, and puttest.py.

LinkXall's version control provides an example of the kinds of sub-projects that are appropriately stored under unique version names in a shared repository. However, because dv.py and bv.py don't require any setting up, it is easy to develop each sub-project in its own directory with its own version control. In fact, the overall LinkXall project includes several of these and, although they are separate, they still need some coordination. This topic is too complex to be discussed here but you can read about these workspaces in the LinkXall design document. If you have installed LinkXall the topic is [LxaDesign.odt*DevelopmentWorkspaces].

A project in its own directory with its own PVC repository can be transformed into a sub-project in another directory by copying its files into the other directory and its bak directories into the other directory's bak. However, if you accepted the default v+ version names, these will probably have to be renamed because only one version group can have this name. The inverse transforms a sub-project into an independent project and its version directories don't need to be renamed.

The automatic incremental version naming simplifies putting a version and seeing the order of versions. However, the alphabetic order of version names doesn't match their time order. For example, in alphabetic order v2 appears after v10. If the repository contains no sub-groups then dir /od in Windows and ls -t in Linux will show the version directories in time order. If it contains sub-groups this would mix the groups, making the list difficult to decipher. PVC address this with the -v operation, which list the sub-groups separately with their versions in time order.

To top of [USE]

COMMAND LINE

[Topic End] [Topics]
[bv.py*CommandLine] [dv.py*CommandLine] Only LinkXall can follow these hyperlinks.

The canonical command line for both dv.py and bv.py is VERSION FILES OPERATION OTHER EXT. If a Configuration file defines VERSION or FILES, these don't have to be included in the command line but if they are they will override their configuration file definitions. Except for OPERATION, whose options are syntactically unique, the role of each argument is determined strictly by its relative position in the command line. VERSION is normally first but, if a configuration file defines VERSION then OPERATION may be first. FILES can only follow VERSION and precede OPERATION. OTHER can appear only immediately after OPERATION and EXT only after OTHER. If OPERATION is blank, OTHER and EXT would be interpreted as FILES elements. If OTHER is blank, EXT would be interpreted as OTHER.

-V is a special case that takes no other arguments and doesn't read or write any repository files. It shows the repository versions (directories) in alphabetic order with each group in temporal order. This is smarter than simple alphabetic order, which would list, for example, file10 before file2, and simple temporal order, which would provide no indication of order within individual groups. This command is usually used to find the most recent version of a particular group without doing anything to it.

<VERSION>

VERSION is required in all cases. It specifies a version name, which must refer to an existing version for all operations other than Put. VERSION is normally a simple name, which identifies the directory under CWD/bak. However, it may include a path to specify an AlternateRepository.

VERSION may specify an exact name or a pattern used to derive a name. The pattern is not globbing or regular expression but a base name with last character "+" which means to derive the name from the most recent version indicated by base name. If base name is blank (VERSION is simply +) the most recent version, regardless of name, is selected. Otherwise, base name may be any initial fragment of a version group. For example, if the version group is dvc, VERSION may be dvc+, d+, or dv+ as long as no other version groups fit the pattern. The repository is searched for the most recent version fitting the pattern. For example, if one group is dvc and another dvctest, the most recent version in either may be selected by d+. In this case, dvct+ will always refer to the dvctest group but dvc+ may refer to either group. Using an exact name for VERSION eliminates this ambiguity but requires manually examining the repository to determine the most recent existing version. Routine use can be simplified by giving each group name a unique first letter.

If VERSION is incremental ("+" suffix) the most recent version passing the base name filter is used for all operations other than put.

<FILES>

FILES comprises one or more arguments (delimited by space) specifying the files involved in the operation. Most commonly FILES tells which files in CWD to Put into a new version. The dv.py Get operation also accepts FILES to limit the files copied from a version into another directory. Each may name a specific file or a group pattern, with "*" specifying all files in CWD, "*.ext*" all files with *ext* extension, and "." all files with no extension.

Sometimes we want to put all of particular file type with a few exceptions. Specific files can be excluded from the put operation with the FILES argument -[*file1name*,*file2name*,etc]. This must appear in the command line before the general argument that would include the files. For example, the LinkXall project workspace includes two types of documents, three that explain its operation and many

used for testing. To divide these into two sub-projects, the command dv.py doc+ LxaUserGuide.odt LxaDesign.odt LxaTut.odt puts just these three files into a doc version while the command dv.py test+ -[LxaUserGuide.odt,LxaDesign.odt,LxaTut.odt] .odt puts all of the others into a test version. If the two sub-projects evolve at different rates, this can reduce memory waste with bv.py, which copies all files. This is not an issue with dv.py but separating files into groups can minimize complexity when restoring and comparing versions.

<OPERATION ARGUMENT>

The OPERATION argument specifies the operation. To reduce the possibility of a file pattern in FILES being interpreted as the OPERATION, OPERATION options have names that would not be desirable for files. They are -PUT, -P, -CMP, -C, -LST, -L, -SRC, -S, -REF, -R, -GET, -G, -DEL, -D in dv.py and -PUT, -P, -CMP, -C in bv.py. -PUT/-P is supported as an OPERATION only for consistency. If the command line includes no OPERATION argument, put is assumed. bv.py supports only put and compare because the other operations exist only to simplify interpreting the delta-based repository of dv.py. [Put] [Cmp] [Lst] [Src] [Ref] [Get] [Del]

<OTHER ARGUMENT>

For all operations except src the OTHER argument specifies a version (other than VERSION) or a directory (by absolute or CWD-relative path). Blank or "." OTHER selects the project directory (CWD). If OTHER names an existing version, it is always interpreted as a version, i.e. a repository directory. For example if both version pvc2 and directory CWD/pvc2 exist, if the OTHER argument is pvc2, it will be interpreted as the version (in the repository). Although not enforced, if OTHER names a version it should be in the same group (base name) as VERSION. It can be just a number, which is applied to the base name indicated by VERSION.

<EXT ARGUMENT>

The EXT argument serves different purposes depending on OPERATION. With Cmp it names a file in VERSION and OTHER to be compared in detail. With Del it names a directory into which unreferenced files are copied before their version directory is deleted.

<@CLAUSE>

To avoid irreversible command line mistakes, both bv.py and dv.py ask before doing something that will change files. For testing, the response to these queries needs to be verified. Automating the response ensures that the tests perform predictably, simplifying regression testing. These queries are done though a function that either asks the user or retrieves the response from the command line. To support this, a special clause may be appended to the command. It is @ followed by any number of Y or N responses, which are provided in sequence to multiple queries. In most cases there is only one of these. The most complicated case involves dv-del. The command dv.py x0 -del x7 dead @ Y Y N tells to delete x0 through x7 and answer Y to delete the versions, Y to clear the dead directory before copying unreferenced files into it, and N to remove the version directories. If there are more queries than @ arguments, the sequence repeats. For example @ Y answers yes to all queries. The @ clause can be included in a manual invocation of a command but it is intended for regression testing. When using bv.py or dv.py normally or for learning how they work it is more reliable and informative to interactively respond to queries.

OPERATIONS

<PUT>

bv/dv VERSION FILES -put

The put operation (-put, -p, or nothing) copies the files in CWD that pass the FILES filtering into the version directory specified by VERSION. If FILES is blank you will be asked whether you want to include all CWD files in the version. If FILES is "*" all files are included without question. Usually all files in the project directory are closed before putting a version. However, it is sometimes convenient to leave them in an active edit state, in which case the editor may have created a temporary lock file that would always be included in "all" and might be accepted by FILES. For example, if this document (PersonalVersionControl.odt) were open, even if not edited, the file ".~lock.PersonalVersionControl.odt#" would be in the project directory and FILES argument ".odt" would accept it. If bv.py were being edited in Emacs, the file ".#bv.py" would exist and be accepted by the FILES argument ".py". Many other editors also use ".~" and ".#" prefixes for this purpose.

Lock files should be routinely excluded from a version, which the put operation does by asking the accept function (bv.py*accept, dv.py*accept) whether or not to accept a file that passes a FILES pattern. A file that appears to be one of these will be accepted without question if explicitly named by one of the FILES arguments. This enables including in the version a file whose name matches an excluded pattern but is not actually a lock file. To provide lock file filtering for editors that use different schemes, the accept function must be edited, which requires some non-trivial Python programming, notably how to use slice or regular expression to examine the beginning and end of a file name. This can be avoided simply by always closing files with lock companions that might be inadvertently included in a version being put.

If Version is incremental ("+" suffix) the version name is derived by incrementing a numeric suffix of the selected most recent version. If it doesn't already have a numeric suffix, 0 is appended. Otherwise, the entire numeric suffix is incremented without limit. For example, if the most recent doc version is doc99, the next one will be doc100. The operating system will by default show these in alphabetic order, which can be misleading, e.g. showing doc100 before doc99. bv.py and dv.py are not mislead by this because they determine order by version directory modified time.

If the version directory already exists, the put operation will show you all of the files that it contains and ask you if you want to overwrite it and, if you agree (answer "Y") ask whether you want to delete all existing files. If VERSION is incremental this could occur only as a result of actions outside of bv.py and dv.py, but otherwise may occur because you want to change the version. It is reasonable to do this with the most recent version ("tip") when you discover that you put it prematurely and have corrected an error in it. Although doing this with older versions defeats the rationale of version control in most situations, both bv.py and dv.py allow you to do it. In most cases, the version directory doesn't already exist and the put operation creates it.

bv.py and dv.py treat versions differently. When a version is put, bv copies all files into the directory, while dv copies only those that are unique and creates a dictionary file (vdict) that tells the version source for all files in the version. For all other operations, dv examines the dictionary to find the source of each file in the version. For cmp, bv treats a version as an ordinary directory located in the repository.

Both bv.py and dv.py report the files that are put to the version. bv.py lists all files as "copied into" the version directory. dv.py tells "[*oldfiles*] *newfiles* put to" the version directory, where *oldfiles* lists those

that have not changed and are only referenced (in vdict) by the version and *newfiles* lists those that are unique and copied into the directory.

<CMP>

bv/dv VERSION -cmp OTHER EXT

The cmp operation (-cmp or -c) compares the files that comprise VERSION to those of another version or directory specified by OTHER. By default, cmp compares all files in VERSION and OTHER and only reports whether they are the same, different, or missing (in one but not the other). It doesn't tell the details (other than age) of how two files differ. The EXT argument changes this behavior. If it is not blank, cmp opens an external file compare program, such as Winmerge or Meld, passing the VERSION and OTHER instances (by path) of the file named by EXT. The program selected for this is hard-wired into bv.py and dv.py in the function detcmp [bv.py*detcmp] [dv.py*detcmp]. Changing the program can only be done by editing these scripts but little understanding of Python is needed. For example, Meld is invoked by os.system('meld %s %s'%(f1,f2)) if it is in PATH. Replacing meld with another program or full path is obvious. Both scripts provide several alternatives as comments. It should be noted that general-purpose file compare programs are only useful for plain text. Documents are compared using the program that creates them (e.g. MS-Word or LibreOffice). To compare a file in the project directory OTHER must be "." because if it were blank, EXT would be interpreted as OTHER.

When bv.py and dv.py compare multiple files they issue similar summary reports that group files by their compare status. Files determined to be identical by examining their content are reported in the "Match" list. This is the only way that bv.py can compare files. dv.py always examines the VERSION dictionary to find the source of its files. If OTHER is a version its dictionary is also examined. If both refer to the same source for a file, it is included, with source version prepended, in the "Same file" list. For example, if versions x4 and x5 are compared and both refer to file 2.x in version x0 then x0/2.x is included in the "Same file" rather than "Match" list. This purposefully simplistic example is from the dv regression test script testcmp.py. Files that are in VERSION but not OTHER are reported in the "Missing" list. The "In *other* but not in *version*" list reports the inverse. If OTHER is a directory it may contain many files that are not part of VERSION. To avoid cluttering the summary this is the last group in the report. Files that are different are reported in the "Mismatch" list, each on a separate line because these are reported with more details. The files are compared by content but whether the VERSION file is older or newer than the OTHER file is reported. Also, to be clear about the relationship, each file is reported with its version prepended. For example (from testbv.py) bv.py reports "bx6/1.x older than bvtest/x666/1.x" when comparing version bx6 and directory bvtest/x666, which contain mismatched 1.x files. dv.py further complicates the report by telling the source of a referenced file. For example (also from testcmp.py) dv.py may report "x6>x0/2.x older than gettest/x666/2.x" when comparing version x6, which refers to the 2.x file in version x0, and directory gettest/x666, whose 2.x file is different. This extra information can be helpful in special circumstances but is not essential to further investigation. The command dv x6 -cmp gettest/x666 2.x opens the external compare program to compare the two files in detail.

<SRC>

dv VERSION -src

File sources in a version created by bv.py are obvious since all of the files are copied into the directory. In versions created by dv.py only unique files are copied into the directory. dv.py provides commands that enable most operations without you having to know the physical location of files that comprise the version, but it is sometimes helpful to know the source of a file. A version's dictionary file provides this information but interpreting it directly requires knowing where it is and understanding its syntax, which

is optimized for performance rather than human-readability. The command dv VERSION -cmp translates the dictionary to a more readable list of the source of every file in the version. Its report uses the same syntax as cmp, for example (from testcmp.py) dv x6 -src might produce:
x6 sources are x5/10.x, x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x6/1.x, x6/11.x.
OTHER and EXT are always blank (ignored) if the operation is src.

<LST>

dv VERSION -lst

dv VERSION -src lists all of the files that comprise VERSION with their version source. This is informative but excessively complex if the user only wants to know what files comprise VERSION. The -LST operation reduces the display to only the file names without their source.

<REF>

dv VERSION -ref OTHER

Ref is the inverse of src, telling which, if any, other versions in the group indicated by VERSION refer to files in VERSION. If OTHER is not blank (number suffix applied to VERSION or a complete name in the group indicated by VERSION) references to files in all versions between VERSION and OTHER are reported. Only files that are unique to the specified versions are analyzed for references, i.e. ref does not report references to references. A manual inspection of the versions could reveal unique files that are not referenced by any other versions but ref simplifies finding such files by reporting these "Unreferenced" files. The "Unreferenced" list is usually empty unless OTHER specifies the most recent group version because few unique files exist in any other version. In all cases, referenced files are reported by *version/file* < *versions* where *version* is where *file* resides and *versions* is a list of all versions that refer to it. When OTHER is blank, *version* is always VERSION but including it in the report makes the situation clear. For example, x0/2.x < x1,x2,x3,x4,x5,x6,x7 tells that versions x1, x2, x3, x4, x5, x6, and x7 refer to the 2.x file in version x0.

<GET>

dv VERSION -get OTHER EXT

The get operation copies the files that comprise VERSION into the OTHER directory. OTHER is never interpreted as version name. If it coincidentally names an existing version, it is not interpreted as that version in the repository but as a directory of that name in the project directory and, if it doesn't already exist, is created. OTHER may be blank or "." to indicated CWD. Optional EXT can name a specific file or extension (e.g. ".x" or "." for no extension) and no other files are copied. In this case, CWD OTHER must be "." as blank would cause EXT to be interpreted as OTHER. For example, dv VERSION -get . . would copy to CWD all of the files without extension in VERSION.

Get would be superfluous to bv.py because standard file copy could accomplish the same thing although with a bit more user effort. If the directory doesn't already exist, dv.py automatically creates it, whereas standard file copy would concatenate all of the files into one file. If the directory already exists and contains files, dv.py asks whether to delete all of the files before copying the version. Get affords more than these modest conveniences. Most version directories contain relatively few actual project files. To get all of the files specified in a version, its dictionary has to be interpreted and many of its files copied from earlier versions. This would be quite complicated and time-consuming to do manually. dv.py does it automatically and practically at the same speed as simply copying the files from a single directory. However, the process is deliberately not entirely transparent because it can sometimes be helpful to know the file sources, which dv.py reports as it copies each file. For example, the testget.py test script invokes dv.py with the command dv x2 -get gettest/x2 to copy all of the files in

the x2 version to CWD/gettest/x2. As it copies the files, dv reports x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/1.x, and x2/7.x, indicating that x2 directly contains only 1.x and 7.x while referring to version x1 for files 5.x and 6.x and to version x0 for files 2.x, 3.x, and 4.x.

Get is usually used to restore a version to the project directory, especially when an apparently new problem occurs. In this case it is advisable to record (put) a new version before overwriting the current project files. This version may have a completely different name, e.g. temp+, from the sub-project (group) under development but it might also be the next group name, e.g. dvc+. If you subsequently decide that it contains nothing of interest, you can replace the version by explicitly naming it in a put command or by using the del command to delete it.

dv.py commands transparently provide the usual capabilities for managing versions regardless of where their files are actually located. General-purpose programs and utilities can't interpret its dictionaries yet may occasionally provide a useful capability not supported by dv.py, for example to export a particular sub-project to a remote location for backup or sharing. The get command affords a workaround by copying all of a version's project files to one directory, which can then be manipulated by other tools. It is possible to get multiple sub-projects into one directory by declining to delete its existing files but this would not usually be useful. The most likely reason for such a snapshot of all groups would be to export the current version of every file in all groups. These are automatically located in the project directory, which may also contain some superfluous files but these can be ignored or deleted.

<DEL>

dv VERSION -del OTHER EXT

In the typical "rolling rev" version control scheme implemented by bc.py, old versions are irrelevant to newer ones because they contain only files that are either duplicated or replaced in newer versions. Traditionally, older versions are manually deleted and version numbering rolls over. bv.py supports this but without any specific functions. In a bv repository, the old version directories are manually deleted and the next bv-put VERSION explicitly restarts group numbering. The delta scheme of dv.py is much more efficient. Old versions don't contain any files that are duplicated in newer versions. Deleting versions is not needed to save storage space but might be done to clean up the repository or to implement a rolling rev to limit the sequential naming range. Manually deleting versions created by dv.py would be very difficult. Files in deleted versions referenced by later versions must be moved to the earliest undeleted versions that reference them and all dictionaries updated to reflect this. dv-del automates this complex process.

Usually a range of versions, VERSION through OTHER, is deleted. OTHER must be in the same group as VERSION. Although OTHER may be a complete version name, it may be just a number, which is combined with the group identified by VERSION. For example, dvc0 through 10 is interpreted as dvc0 through dvc10. Sometimes a single version at any level of the group is deleted. This is not done to reuse the version name but to purge the repository of a bad version, whose presence might be misleading. To delete one version, OTHER is blank or the same as VERSION.

Most files in deleted versions are referenced by later versions and are saved by moving. Files that are not referenced at all or only by other versions being deleted are obsolete but may still have some useful content. If EXT is not blank it names a directory, into which all of these "orphan" files are copied before their versions are deleted. Many of these may be different versions with the same name. To avoid conflict and to clarify where they all came from, the version name is prepended to every one of these saved files. For example (from testdel.py) dv.py x0 -del x7 dead copies the unreferenced files in

deleted versions x0 through x7 into CWD/dead, renaming them x0-1.x, x0-2.x, x0-3.x, x0-4.x, x1-5.x, x1-6.x, x2-1.x, x2-7.x, x3-8.x, x4-1.x, x4-9.x, x5-10.x, x6-1.x, x6-11.x. If only one version is deleted OTHER must repeat VERSION. If it were blank EXT would be interpreted as the last deleted version.

Without actually deleting versions there is no automated means of finding the files that would become obsolete if later versions were deleted. Ref shows as "Unreferenced" only the files that are not referenced by any other version. However, for all referenced files it lists the referring versions. These lists can be manually inspected to find files that are only referenced by versions being considered for deletion.

Before deleting versions, dv asks you whether that is what you really want to do. Deleting versions copies their referenced files to versions that will not be deleted and (optionally) unreferenced files to EXT. However, the version directories and their files are removed only after asking you if that is what you want to do. You may want to keep them for reference but they no long play any part in the repository. Removing them not only reduces storage waste but also clarifies the repository by removing superfluous directories. If you elect to remove these directories dv will ask if you want to shift the rest down to fill the gap. Doing this further clarifies the repository and, like a rolling rev, limits version name growth. However, it is better than a rolling rev, whose wrap-around confuses the relative age of versions indicated by their name. Before copying unreferenced files to EXT, you are asked whether you want to delete files currently in EXT. Keeping the files enables accumulating the obsolete files from multiple independent version deletions in the same or different groups.

If you delete versions without removing their directories and subsequently want to shift the later ones down to fill the gap, you can repeat the delete command and choose both directory removal and shift. If you delete versions and remove their directories but decline shift there will be no simple way to subsequently request it because dv calculates the name change based on the deleted range. However, you can prepare for this manually by making the missing group version directories. These can be empty because none of the later ones will still be referring to them. You could also use this to downshift an isolated group. For example, to move the group x4,x5,x6 down to x0,x1,x2, which don't exist, you can make empty directories x0,x1,x2,x3 and then invoke dv x0 -del x3 and elect to remove the directories and shift the rest.

To top of [COMMAND LINE]

CONFIGURATION

[Topic End] [Topics]
[bv.py*Configuration] [dv.py*Configuration]

Most version control programs have both a command-line and a GUI version. The GUI is not very efficient for routine use but can simplify configuration with specialized dialogs. Because bv.py and dv.py are open source Python scripts, their behavior can be easily modified by anyone with even a modest understanding of Python. Consequently, they don't need the extra complexity of a GUI shell or command line configuration arguments. It would be feasible to simplify routine use with project-specific versions but bv.py and dv.py provide a more convenient means of achieving this with a project-specific configuration file in the project directory (CWD). For bv.py this is bvcfg and for dv.py it is dvcfg. Both of these can potentially define three things, the repository directory (REPDIR) and the default command arguments VERSION and FILES. Any line that begins with # is a comment.
For example, in the personal version control project directory, bvcfg contains:

#REPDIR=? VERSION=? FILES=?
VERSION=bvc+
FILES=.py
and dvcfg contains:
#REPDIR=? VERSION=? FILES=?
VERSION=dvc+
FILES=.py
For both bv.py and dv.py, put is the default operation. Invoking either without arguments captures the current version of all Python project files in the next version of bvc or dvc. This is the most common operation. For all other operations, bvc+ or dvc+ is interpreted as the most recent version of the group (bvc or dvc). Invoking bv.py or dv.py with just the operation argument -cmp (or -c) compares the files in the most recent bvc or dvc version to those in the project directory (CWD). The EXT argument is also supported. bv -c . *file* and dv -c . *file* invoke the external file compare program to compare *file* in the most recent bvc or dvc version to CWD/*file*. "." in the command line, which means to compare to CWD, is required to avoid *file* being interpreted as OTHER.

The VERSION and FILES arguments defined by bvcfg and dvcfg simplify routine use but can be overridden by invocation command-line arguments. There is no specific REPDIR command line argument but if the VERSION argument includes path, the path part serves as REPDIR and the remainder as VERSION. Both REPDIR and VERSION may specify the repository path as absolute or relative (to CWD). For example, if VERSION is dvc0 and REPDIR is not defined, the version directory is CWD/bak/dvc0. If the configuration file defines REPDIR as dvbak, the directory is CWD/dvbak/dvc0. If VERSION is rep/dvc0, the directory is CWD/rep/dvc0. A repository defined by VERSION also applies to an OTHER argument, unless OTHER includes path, which causes it to be interpreted as a non-version directory.

Neither the command line nor the configuration file can choose the program invoked for a detailed (and usually interactive) comparison of two files. This is hard-wired into the detcmp function (bv.py*detcmp, dv.py*detcmp). Changing this requires editing the script but little knowledge of Python. In most cases, it only requires changing the name of the program. However, if it is not in PATH, the full path to the executable is needed. Also, some programs may need to be invoked through the shell, which requires the files to be identified by an absolute OS-specific path. detcmp includes a commented example of this.

<MINIMAL CONFIGURATION>

It is not difficult to enter a full command line and the configuration file is not complicated. However, we sometimes want to immediately begin version control on a new simple project without being distracted by version control considerations. If bv.py or pv.py is invoked for the first time without arguments and there is no configuration file, after confirming that this is your intent, they will make the version directory bak/v0 and copy all files in the project directory into it. If invoked repeatedly without arguments they will create sequential version directories v1, v2, etc, always asking whether this is your intent and warning if none of the files have changed. If invoked with just the -cmp or -get OPERATION, they will compare the most recent version or copy it to the project directory. Initiating version control in this manner does not restrict operations in any way.

<ALTERNATE REPOSITORY>

The alternate repository, defined by REPDIR in the configuration file or embedded in the VERSION argument, has a wide variety of uses. In the simplest case, it completely replaces the default CWD/bak

in order to make the repository independent of the project directory. This might be done to put the repository on another drive for automatic backup or to make it accessible to other systems.

The alternate repository mechanism might also be used to support separate sub-project repositories. For example, a programming project might use the default repository for its source code version control while versions of documents in the project directory are stored in a repository shared with other projects. The configuration file can only define one REPDIR and it applies to all files. Consequently, this requires a complete command line in the alternative cases, e.g. dv /doc/projects/thisone/v+ .odt. To simplify routine use, this should be implemented by a script.

The alternate repository mechanism can also play an inverse role, not defining an alternate repository but incorporating another into the CWD/bak default. If a project uses bv.py or dv.py for version control from its inception, the repository will contain a continuous record of versions. An existing project or sub-project that comprises only the current file versions in the project directory can simply be put as the first version. However, an existing project not under bv.py or dv.py version control may have older versions recorded in unrelated directories. Manually converting these to a bv repository is relatively simple. Sequential version directories under REPDIR would be created and the specific version files copied into them. Conversion to a dv repository is much more complicated because each version directory contains its unique dictionary file vdict and only project files that are different from the previous version. This can be done by putting each version from its current directory specifying the repository in the VERSION argument. For example, CWD contains plain (bv-like) versions v0, v1, v2. These can be converted to bv or dv versions under bak by going to each directory in turn and invoking bv or dv with VERSION argument ../bak/v+. Of course, the "v" root name can be changed.

The alternate repository mechanism can also support special cases, where the project directory is not independent. For example, Libre Office provides a convenient means of developing user libraries but only in specific project directories. For example, the LinkXall project directory is C:\Users*name*\AppData\Roaming\LibreOffice\4\user\basic\LinkXall in Windows and /home/*name*/.config/libreoffice/4/user/basic/LinkXall in Linux. The repository should not be under these directories because they are automatically created by Libre and may also be automatically deleted. LinkXall is an especially cogent example because it includes sub-projects that are not part of the library. The LinkXall project directory is C:\swdev\libre\linkxall in Windows and /home/usr/work/swdev/libre/linkxall in Linux. In both, the repository is bak under this but the source files are not necessarily in the project directory. The Python script getlib.py copies the library files from the working directory to the project directory for backup and they could be put from there into repository versions. However, the library contains many different files types and FILES would have to name these to ensure that all of them any no other files are included in the library versions. Further, changes to the library or Libre itself may add new file types to the library or remove some that could still be in the project directory and should not be included in a new version. The easiest and most reliable solution is to capture a version by copying all of the files in the working library to an empty directory and from there put the version specifying the normal repository as "alternate". The putlib.py script in the project directory automates this. It creates empty directory "tip" and fills it with the working library files plus the configuration file from the LinkXall user's data directory (which Libre may also delete at any time). The script changes CWD to tip and invokes dv.py ../bak/lib+ * to record the next sequential lib+ version comprising all files in tip. This could also be done with bv.py but the library contains many files that don't change and the efficient storage of dv.py reduces the urgency to delete old versions to recover disk space. However, since the version directories don't contain all library files, restoring a previous version can't be done by simply copying their files back to the working directories. Instead, the reverse script replib.py does this. It takes one argument, which is either "tip" or

specific version name, e.g. lib99. For tip it copies the files from the tip directory. Otherwise, it invokes dv.py *version* -get librep and then copies the files from the librep directory. The putlib.py and replib.py operations could be folded into dv.py but this is a unique case, which can only serve as a model for other situations.

To top of [CONFIGURATION]

TESTING

[Topic End] [Topics]

bv.py has only two operations, put and cmp, which are relatively easily verified by manual inspection because every version directory contains all project files. Verifying dv.py is much more complicated. Not only does it have the additional operations of src, ref, get, and del, but also its delta-based repository cannot be verified by simple file comparison. It can only be effectively verified by automated means. Each of its operations is tested by a specific script, testput.py, testsrc.py, testref.py, testcmp.py, testget.py, and testdel.py. These scripts not only verify the operations but can be used for regression testing of any changes to dv.py. Although not as critical as for dv.py, testbv.py provides regression testing of both put and cmp operations of bv.py. These test scripts also provide useful instruction with examples of many different ways to use the functions provided by dv.py and bv.py.

Put, get, and del operations modify repository and/or other directories and can be verified by comparing the results to what is expected. Cmp, src, and ref don't modify any files but only display status for the user to examine. These operations are verified by redirecting displays into temporary files, which are then compared to what is expected. To use these tests for learning about bv.py and dv.py, you can manually invoke the commands and inspect the results. Each test is described here not only to justify its operation but also to enhance its instructional value.

Tests require specific project directory and repository conditions to be able to compare actual to expected results. Put is the only operation that builds the repository, which must first be done before testing any of the other operations. testbv.py tests put and then cmp, ensuring that the cmp tests are provided with a consistent project directory and repository. Testing each dv operation with an independent script simplifies and accelerates development but opens the possibility of one test script changing the initial conditions of the next script to execute. In particular, testdel.py changes the repository, testget.py changes test files in the project directory, and testcmp.py adds to the repository some test files that don't exist immediately after testput executes. Therefore, all of the dv test scripts (except, obviously, testput.py itself) include an initial invocation of testput.py. However, this can be commented out to save time if the same script executes repeatedly or the scripts execute in the order, testput, testsrc, testref, testcmp, testget, testdel.

The test scripts invoke bv.py and dv.py with complete command lines and assume the default repository (CWD/bak) making bvcfg and dvcfg irrelevant.

TEST PUT

testput.py and the put section of testbv.py [testbv.py*Put] create identical repository groups (under bak) x0 through x7 for dv and bx0 through bx7 for bv. This is coincidentally an example of how dv and bv support multiple sub-projects in one project directory. To avoid test condition carryover, testput.py and testbv.py initially delete all .x files in the project directory. testput.py removes all x

version in the repository and testbv.py removes all bx versions. They then create the same set of test files 1.x through 11.x and put them to the versions x0 through x7 or bx0 through bx7.

Very simple text files 1.x, 2.x, 3.x, and 4.x (containing respectively 1x, 2x, 3x, 4x) are created in the project directory. testput.py puts these to version x0 with the command dv.py x+ .x. testbv.py puts them to version bx0 with bv.py bx+ .x. Since all group versions have been deleted, dv.py translates x+ to the new version x0 and bv.py translates bx+ to bx0. Similarly simple 5.x and 6.x are created and put to x1 and bx1 with the commands dv.py x+ .x and bv.py bx+ .x. Since x0 and bx0 now exist, x+ and bx+ are translated to x1 and bx1. dv.py copies only the new files 5.x and 6.x to x1 and adds the unchanged 1.x through 4.x to the x1 dictionary (vdict). bv.py copies all of the .x files into bx1. Similarly 7.x through 11.x are created and put to x2 through x6 or bx2 through bx6 with the same command dv.py x+ .x or bv.py bx+ .x. Simultaneously, a unique 1.x is created for x2, x4, and x6. Thus, x0 and x1 have one version of 1.x, x2 and x3 another, x4 and x5 another, and x6 has a unique version. This is easily verified for bv by comparing directories bv0 through bv6 because bv.py put copies all files to the version directory. Although testput.py creates the same test files and versions as testbv.py, the versions can only be compared by inspecting their dictionaries.

testbv.py and testput.py create one more version, bx7 or x7, using the command bv.py + .x or dv.py + .x. This is intended to test two things. VERSION + means the next version after the most recent regardless of group base [CommandLine]. Since this command executes immediately after bx6 (or x6) version is created, it should make bx7 (or x7). Additionally, since no project files are changed, the user is asked whether they want to make the version anyway. Once this has been verified, @ Y is appended to the command to automatically make the version without asking in order to simplify regression testing.

Other operation tests depend on the put test to establish initial conditions, which are obvious for bv because every version contains all project (.x) files. This is essentially the same initial condition for dv because all of its operations understand how to interpret a version's dictionary to identify the files that comprise the version, including those that reside in an ancestor.

Although AlternateRepository is usually used in a relatively complex manner, for example putting a new version in the default CWD/bak repository from a different directory, testput.py and testbv.py test it with a simple use-case. From CWD they put *.x files into a VERSION specified with a path, dv.py dvalt/x7 .x and bv.py bvalt/x7 .x. It this works, more complex and absolute paths will also work.

<BV PUT TEST>

Verifying the bv-put operation is a simple matter of verifying the files in each version and comparing 1.x files, which is the only project file whose content changes as the versions are being put. The testbv.py*testver function compares a bx version directory to a list of expected files. It is invoked to verify that bx0 contains 1.x, 2.x, 3.x, and 4.x; bx1 adds 5.x and 6.x; bx2 adds 7.x; bx3 adds 8.x; bx4 adds 9.x; bx5 adds 10.x; bx6 adds 11.x; and bx7 contains the same files as bx6.

That the differing versions of 1.x are correctly put is tested by comparing the file in successive versions. It is expected that bx0 = bx1, bx1 != bx2, bx2 = bx3, bx3 != bx4, bx4 = bx5, bx5 != bx6, bx6 = bx7, i.e. that the compare sequence produces alternating True and False.

<DV PUT TEST>

The put operation is tested by [testput.py]. Although dv.py supports defining a remote repository [dv.py*Defaults] testput.py hard-wires the default "bak" under the development directory. It creates a fresh repository group x0 through x6 with files 1.x through 11.x. These are short text files designed to simplify inspection. Initially removing any residual group versions (in bak) and x files (in CWD) ensures that consistent results, which it compares to expected patterns. It not only demonstrates and verifies the put operation but also provides a consistent starting point for testing other operations. The get, compare, and delete operations all modify the repository. To afford an opportunity to review the put test results, testput.py doesn't test any of these itself. Other scripts do that and invoke testput.py themselves to prepare for testing. dv.py executes put operations without asking the user for approval because put only adds to the repository. The other operations modify the repository and/or other directories and, therefore, dv.py tells the user what it intends to do and asks for approval.

To avoid carryover from previous testing, testput.py initially deletes all .x files in the development directory and removes x directories under bak. It then creates new 1.x, 2.x, 3.x, and 4.x text files containing, respectively, 1x, 2x, 3x, and 4x. It invokes dv.py with the command line dv.py x+ .x. An OPERATION argument is not needed because put is the default. The x+ VERSION argument tells to give the new version the next sequential name after the most recent x directory. If there are none of these, dv.py automatically names the version x0. Having no preceding versions in the x group, dv.py copies all four of the x files into x0 and creates the version dictionary file (vdict in x0) containing {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"]}.

testput.py then writes files 5.x and 6.x containing, respectively, 5x and 6x, and repeats the dv.py x+ .x command. dv.py creates the directory x1 under bak and copies these files into it. It writes x1's vdict as {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"]}. This shows that 5.x and 6.x reside in x1 but the earlier files are located by reference to x0.

testput.py next creates the new file 7.x and changes 1.x contents to "1-7-x" and repeats the dv.py x+ .x command. The resulting bak/x2 directory contains the new 7.x and unique 1.x but all other project files are referenced to previous versions. Its vdict is {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"]}. The 1.x CRC 1475035452 no longer matches the 3646614595 of 1.x in x0. 2.x, 3.x, and 4.x are still referenced to version x0. 5.x and 6.x are referenced to version x1 (of course with the same CRC). The new 1.x is after 7.x in the dictionary because testput purposefully creates it after 7.x in order to demonstrate and test that the dictionary is ordered not alphabetically but by file (modified) date. testput creates a new 1.x for every other version and repeats this pattern of making an entirely new file before a modified 1.x.

testput.py creates the new file 8.x and repeats dv.py x+ .x, which creates version x3. 1.x is not changed so bak/x3 contains only the new 8.x. The x3 vdict is {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"], "8.x": [144201482, "x3"]} referring to x0 for 2.x, 3.x, and 4.x, to x1 for 5.x and 6.x, and to x2 for 1.x and 7.x. It refers to itself for 8.x.

For x4, testput creates the new file 9.x and changes 1.x to "1-9-x" so bak/x4 contains 9.x and 1.x. The x4 vdict is {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"]}.

For x5, testput creates the new file 10.x but doesn't change 1.x so bak/x5 contains only 10.x. The x5 vdict is {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"], "10.x": [1534105928, "x5"]}.

For x6, testput creates the new file 11.x and changes 1.x to "1-11-x" so bak/x6 contains 11.x and 1.x. The x6 vdict is {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "10.x": [1534105928, "x5"], "11.x": [1114351625, "x6"], "1.x": [1417620892, "x6"]}.

Because x7 has no new or changed files, its directory should contain only the vdict dictionary file, which is identical to the x6 dictionary. Normally, dv-put won't create a duplicate version without first asking the user, but testput avoids this by creating x7 with the command dv.py + .x @ Y, which tells to automatically answer Y to the query.

[dv.py*Put] creates an actual dictionary from the selected files while examining the most recent version dictionary for matches. It then invokes json.dump to write this dictionary into vdict as a single line formatted as json does a dictionary. testput verifies each put by passing a list of the files expected in the version directory and a copy of the json-formatted dictionary to testput.py*testver, which compares the list argument to the sorted dirlist of the version and the dictionary argument to the string read from vdict.

TEST CMP

The cmp operations of dv.py and bv.py are functionally similar. They compare a version to another version or directory. In the command line dv/bv VERSION -cmp OTHER *file*, VERSION can only be a version name. OTHER can be a version name or an absolute or relative directory, with blank or "." interpreted as CWD. testcmp.py and testbv.py test all variations of VERSION (in bak/x and bak/bx, respectively) but they don't test operation with non-blank *file* argument. In this case dv.py and bv.py invoke another program, which usually implements an interactive GUI, leaving no predictable changes that could be tested. Other compare operations also don't change files or directories, leaving no directly testable results. Instead, they display results to the user. However, these can be tested by redirecting the output into a temporary file, which is then compared to the expected output. bv.py*Cmp and testcmp.py use similar functions testbv.py*testcmd and testcmp.py*testcmd for this. They invoke bv.py or dv.py with the arg argument, which is VERSION -cmp OTHER, redirecting the output into temporary file cmpres, which they then compare to the expect argument and report as "correct" or "wrong".

Except for "Same file" (files that are in a version but only by reference to an earlier version) dv-cmp and bv-cmp display the same things, files that match, files that don't match, files that are in VERSION but not OTHER, and files in OTHER but not VERSION. The latter is displayed last because if OTHER is a directory it may contain many non-project files that would clutter the display. The first display line tells the absolute addresses of VERSION and OTHER. If this were included in the test results, testing would have to be done in a specific directory. To avoid this, the testcmd function (in both testcmp.py and testbv.py) and the expected results list passed to it skip the first output line.

dv-cmp and bv-cmp themselves make no file system changes. Most of the tests performed by testcmp.py and the cmp section of testbv.py are based on the repository as established by testput.py and the put section of testbv.py. However, more thorough testing requires modifying files in the project and version directories. Since testbv.py always does the put testing before cmp, these changes have no effect. Being divided into separated scripts for each operation, testing dv has to be done in a particular order or testput.py executed before each of the others to avoid a mismatch between the correct and expected results.

The operation of bv-cmp is self-evident because all of the files actually reside in selected version directories. dv-cmp operation is less obvious because many of the files involved are only referenced by version dictionaries. testcmp.py and testbv.py don't need to adapt to this difference because dv-cmp and bv-cmp interpret the same command line as appropriate for the repository version group. However, the expected results are not identical because, in addition to the "Same file" list, dv-cmp displays the owner of each referenced file in the "Mismatch" list. The expected results embedded in testcmp.py and testbv.py are simply copied from verified operation displays, automatically incorporating syntactic differences.

<BV CMP TEST>

The first seven tests in testbv.py*Cmp operate on bx0 through bx6 created by the preceding put section and don't make any file system changes.

bv.py bx0 -cmp bx1 should produce ['Match: 1.x, 2.x, 3.x, 4.x\n', 'In bx1 but not in bx0: 5.x, 6.x\n'] because 1.x, 2.x, 3.x, and 4.x don't change between these versions but 5.x and 6.x are created for version bx1. The inverse command bx1 -cmp bx0 should produce identical output. The two commands are conceptually slightly different in that the list of files in OTHER but not in VERSION is always the last section of the display. That is not apparent in this case because bx0 does not contain any files that are missing in bx1 and Match and Mismatch lists are always displayed before Missing lists.

bv.py bx4 -cmp bx0 should produce ['Match: 2.x, 3.x, 4.x\n', 'Mismatch: bx4/1.x newer than bx0/1.x\n', 'In bx4 but not in bx0: 5.x, 6.x, 7.x, 8.x, 9.x\n'] because 2.x, 3.x, and 4.x don't change between versions bx0 and bx4 but bx4 has a new version of 1.x and new files 5.x, 6.x (introduced at bx1) 7.x (introduced at bx2) 8.x (introduced at bx3), and 9.x (introduced at bx4).

bv.py bx0 -cmp bx4 should produce ['Match: 2.x, 3.x, 4.x\n', 'Mismatch: bx0/1.x older than bx4/1.x\n', 'In bx4 but not in bx0: 5.x, 6.x, 7.x, 8.x, 9.x\n'] which conveys the same information as its inverse bx4 -cmp bx0 but Mismatch displays bx0/1.x older than bx4/1.x instead of bx4/1.x newer than bx0/1.x.

bv.py bx5 -cmp bx4 should produce ['Match: 1.x, 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x\n', 'In bx5 but not in bx4: 10.x\n'] because the only difference between bx5 and bx4 is the introduction of the new file 10.x.

bv.py bx5 -cmp bx6 should produce ['Match: 10.x, 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x\n', 'Mismatch: bx5/1.x older than bx6/1.x\n', 'In bx6 but not in bx5: 11.x\n'] because bx6 has a new version of 1.x and the new file 11.x

bv.py bx6 -cmp bx5 should produce ['Match: 10.x, 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x\n', 'Mismatch: bx6/1.x newer than bx5/1.x\n', 'In bx6 but not in bx5: 11.x\n'] which conveys the same information as

its inverse bx5 -cmp bx6 but Mismatch displays bx6/1.x newer than bx5/1.x instead of bx5/1.x older than bx6/1.x.

Further bv-cmp testing requires file system changes. In all preceding tests, OTHER is a version. To test OTHER as directory obviously requires creating a directory with some relationship to the repository. Standard file system functions are used to to create OTHER directories because bv.py has no equivalent of dv-get.

In the first of these tests, the directory CWD/bvtest/bx6 is created by copying the bak/bx6 version tree to bvtest/bx6. Then bv.py bx6 -cmp bvtest/bx6 should produce ['Match: 1.x, 10.x, 11.x, 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x\n']

In the next test bak/bx0 is copied to bvtest/bx0. Then bv.py bx6 -cmp bvtest/bx0 should produce ['Match: 2.x, 3.x, 4.x\n',
'Mismatch: bx6/1.x newer than bvtest/bx0/1.x\n', 'In bx6 but not in bvtest/bx0: 10.x, 11.x, 5.x, 6.x, 7.x, 8.x, 9.x\n'], which is essentially the same result as comparing version bx6 to bx0 except that this tests OTHER as directory instead of version.

For the next test, version bx6 is copied to bvtest/x666 and three new files called one, two, and three are added to x666. This is not significantly different from the introduction of 5.x and 6.x at bx1, 7.x at bx2, 8.x at bx3, and 9.x at bx4, but, in this case, OTHER is a non-version directory. Then bv.py bx6 -cmp bvtest/x666 should produce ['Match: 1.x, 10.x, 11.x, 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x\n', 'In bvtest/x666 but not in bx6: one, three, two\n'].

In its final test of bv-cmp, testbv.py tests a variety of Mismatch cases. It replaces the 1.x, 2.x, 3.x, 4.x, and 5.x files in bvtest/x666, which were originally copied from version bx6, with new versions. It then invokes bv.py bx6 -cmp bvtest/x666, which should produce ['Match: 10.x, 11.x, 6.x, 7.x, 8.x, 9.x\n', 'Mismatch:\n', ' bx6/1.x older than bvtest/x666/1.x\n', ' bx6/2.x older than bvtest/x666/2.x\n', ' bx6/3.x older than bvtest/x666/3.x\n', ' bx6/4.x older than bvtest/x666/4.x\n', ' bx6/5.x older than bvtest/x666/5.x\n', 'In bvtest/x666 but not in bx6: one, three, two\n']. The Match and Missing sections of this output are easily compared but the Mismatch list is grammatically tricky. As explained in Cmp, the Mismatch list displays each file pair on a separate line. The expected result list passed to testbv.py*testcmd has to be formatted correctly for this.

<DV CMP TEST>

testcmp.py operates on versions x0 through x6 and on directory CWD/gettest and is expected to execute after testput.py. Before conducting its cmp tests it may invoke testput.py although this is normally disabled to speed up regression testing and is needed only if testdel.py executes after testput.py. testsrc, testref, and testget don't disrupt the repository in any way that interferes with testcmp.

dv.py x0 -cmp x1 should produce ['Same file: x0/1.x, x0/2.x, x0/3.x, x0/4.x\n',
'In x1 but not in x0: 5.x, 6.x\n'] because x1 refers to 1.x, 2.x, 3.x, and 4.x in x0 and testput.py creates 5.x and 6.x for x1.

dv.py x1 -cmp x0 should produce the same result as its inverse. Files in OTHER but not VERSION are displayed at the end but swapping them doesn't change the "missing" display because x0 doesn't contain any files not in x1.

dv.py x4 -cmp x0 should produce ['Same file: x0/2.x, x0/3.x, x0/4.x\n',
'Mismatch: x4/1.x newer than x0/1.x\n', 'In x4 but not in x0: 5.x, 6.x, 7.x, 8.x, 9.x\n'] because x4 has 5.x-9.x that don't exist in x0 and its own version of 1..x.

dv.py x0 -cmp x4 should produce ['Same file: x0/2.x, x0/3.x, x0/4.x\n',
'Mismatch: x0/1.x older than x4/1.x\n',
'In x4 but not in x0: 5.x, 6.x, 7.x, 8.x, 9.x\n']
which is conceptually the same as its inverse but the Mismatch report is slightly different because it always shows whether VERSION is newer or older than OTHER, i.e. x0/1.x older than x4/1.x vs. x4/1.x newer than x0/1.x.

dv.py x5 -cmp x4 should produce ['Same file: x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x\n, x4/1.x',
'In x5 but not in x4: 10.x\n']
because x5 has the new 10.x and gets 1.x and 9.x from x4 and all others from the same x0, x1, x2, and x3 as does x4.

dv.py x5 -cmp x6 should produce ['Same file: x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x5/10.x\n',
'Mismatch: x5>x4/1.x older than x6/1.x\n',
'In x6 but not in x5: 11.x\n']
because x5 gets its 1.x from x4 whereas x6 has its own 1.x, and the sources of all other files are the same for both x5 and x6 and only x6 has 11.x. This coincidentally illustrates and tests that dv-cmp displays in mismatches the compare version and source if reference (x5>x4/1.x) and the compare version otherwise (x6/1.x).

dv.py x6 -cmp x5 should produce ['Same file: x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x5/10.x\n',
'Mismatch: x6/1.x newer than x5>x4/1.x\n',
'In x6 but not in x5: 11.x\n']
which conveys the same information as the inverse x5 -cmp x6 but Mismatch shows x6/1.x newer than x5>x4/1.x instead of x5>x4/1.x older than x6/1.x

dv x6 -cmp x0 should produce ['Same file: x0/2.x, x0/3.x, x0/4.x\n',
'Mismatch: x6/1.x newer than x0/1.x\n',
'In x6 but not in x0: 5.x, 6.x, 7.x, 8.x, 9.x, 10.x, 11.x\n']

All of the preceding tests operate on the repository versions created by testput.py and don't make any file system changes. The remaining tests create directories to be used for OTHER and files that don't match the repository.

A simple test of OTHER as directory instead of version is prepared by invoking dv.py x6 -get gettest/x6 @ Y, which creates CWD/gettest/x6 copy of the x6 version. The @ Y clause provides an automated response to the query whether to delete existing files in gettest/x6. When testcmp.py executes for the first time, gettest doesn't exist and the query is not issued. However, none of the scripts remove this so it will exist (with files) the next time testcmp executes even if testput intervenes. Deleting the files ensures no carryover from the previous invocation. Then dv.py x6 -cmp gettest/x6 should produce ['Match: 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x, 11.x, 1.x\n']. Note that, although gettest/x6 contains

all of the files, bak/x6 only contains 11.x and 1.x, the rest being referenced to older versions. dv-cmp includes this detail only in Same and Mismatch lists.

To test VERSION compare to a directory with files that are not identical, the command dv.py x0 -get gettest/x0 @ Y is invoked and then dv.py x6 -cmp gettest/x0, which should produce ['Match: 2.x, 3.x, 4.x\n', 'Mismatch: x6/1.x newer than gettest/x0/1.x\n', 'In x6 but not in gettest/x0: 5.x, 6.x, 7.x, 8.x, 9.x, 10.x, 11.x\n']. This is conceptually the same result as for dv x6 -cmp x0 except that 2.x, 3.x, and 4.x are reported as Match rather than Same file.

To verify that dv-cmp correctly reports non-project files when OTHER is a directory, dv.py x6 -get gettest/x666 @ Y is invoked to create CWD/gettest/x666 containing only the project files of version x6. Then three non-project files called one, two, and three are created in this directory. Then dv.py x6 -cmp gettest/x666 should produce ['Match: 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x, 11.x, 1.x\n', 'In gettest/x666 but not in x6: one, three, two\n']. Unlike project file lists, which are always ordered by modification date, non-project files are listed in alphabetic order, e.g. one < three < two.

The Mismatch list is more complicated than Same, Match, and missing. If there is only one mismatched pair it is displayed on one line with "Mismatch". This is the case within the repository because testput doesn't create any unique files other than 1.x. If there are multiple mismatched files, each pair is displayed on a separate line with file reference details, for example x6>x0/2.x. This is verified by replacing with new versions the 1.x, 2.x, 3.x, 4.x, and 5.x files from x6 in the gettest/x666 directory created in the preceding test. The unchanged files 6.x, 7.x, 8.x, 9.x, 10.x, and 11.x should be reported as matching and the non-project files one, two, and three should be reported as missing in x6. All of the new new file versions should be reported as mismatch with either the x6 version older or the x666 version newer. dv.py x6 -cmp gettest/x666 should produce:
['Match: 6.x, 7.x, 8.x, 9.x, 10.x, 11.x\n',
'Mismatch:\n',
' x6>x0/2.x older than gettest/x666/2.x\n',
' x6>x0/3.x older than gettest/x666/3.x\n',
' x6>x0/4.x older than gettest/x666/4.x\n',
' x6>x1/5.x older than gettest/x666/5.x\n',
' x6/1.x older than gettest/x666/1.x\n',
'In gettest/x666 but not in x6: one, three, two\n']

To verify multiple mismatched project files when OTHER is a version, new versions of 1.x, 2.x, 3.x, 4.x, and 5.x are created in the project directory (CWD) and put to the new version x8, which is then compared to another version (arbitrarily x4). Before doing this, a baseline test gets the unmodified files to x8, with dv.py x8 .x @ Y. @ Y ensures that x8 contains no residual files if testcmp executes repeatedly with no intervening testput, which would delete x8. This coincidentally tests a feature of the @ clause, which is that its argument list is circular if there are more queries than answer arguments. In this case the user would be asked whether to overwrite x8 and whether to delete its existing files. The answer to both is Y. The 2.x through 9.x put to x8 will be sourced by x0 through x4. 1.x and 11.x will be sourced by x6 and 10.x by x5.Then dv.py x4 -cmp x8 should produce:
['Same file: x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x\n',
'Mismatch: x4/1.x older than x8>x6/1.x\n',
'In x8 but not in x4: 10.x, 11.x\n']
This baseline test requires CWD to contain 1.x through 11.x. For the modified 1.x through 5.x test, the original versions of these files are not needed but the other files are. The testget script deletes all .x files

in CWD for one test but restores them so that testcmp can execute after testget without repeating testput.

After making the new versions of 1.x through 5.x in CWD, all .x files are put to a fresh x8 version by dv.py x8 .x @ Y. Then dv.py x4 -cmp x8 should produce:
['Same file: x1/6.x, x2/7.x, x3/8.x, x4/9.x\n',
'Mismatch:\n',
' x4>x0/2.x older than x8/2.x\n',
' x4>x0/3.x older than x8/3.x\n',
' x4>x0/4.x older than x8/4.x\n',
' x4>x1/5.x older than x8/5.x\n',
' x4/1.x older than x8/1.x\n',
'In x8 but not in x4: 10.x, 11.x\n']
As with testget, file system changes made for this test would force repeating testput for some other test scripts to produce the expected results. To avoid this, testcmp invokes dv.py x7 .x -get to restore the .x files. This coincidentally tests dv-get to CWD (blank OTHER). It also removes bak/x8, which could be seen as the most recent x group version by other tests. This would not be a problem for testput [DvPutTest] which initially removes all x group versions in the repository and .x files in CWD.

DV SRC AND REF TESTS

[testsrc.py] [testref.py]

Similarly to dv-cmp, dv-src and dv-ref themselves only display status reports and don't modify the file system. To test them, testsrc.py and testref.py employ the same testcmd function as testcmp.py ([testcmp.py*testcmd testsrc.py*testcmd testref.py*testcmd) which executes the given command, redirecting the display into a temporary file, which is then compared to the given expected results. For generality, testcmp, testsrc, and testref incorporate nearly identical copies of the testcmd function, whose expected results argument is a list of expected output lines. However, only dv-cmp and dv-ref actually produce multiple display lines.

<DV SRC TEST>

dv-src is the simplest dv.py operation. It only involves one version and essentially just translates the version's dictionary into a more human-readable form, displayed as one line. testsrc.py is the simplest tester not only because dv-src is simple but also because the operation can be fully tested using only the repository group created by testput.
dv.py x0 -src should produce x0 sources are x0/1.x, x0/2.x, x0/3.x, x0/4.x
dv.py x1 -src should produce x1 sources are x0/1.x, x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x
dv.py x2 -src should produce x2 sources are x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x2/1.x
dv.py x3 -src should produce x3 sources are x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x2/1.x, x3/8.x
dv.py x4 -src should produce x4 sources are x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x4/1.x
dv.py x5 -src should produce x5 sources are x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x4/1.x, x5/10.x
dv.py x6 -src should produce x6 sources are x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x5/10.x, x6/11.x, x6/1.x
dv.py x7 -src should produce x7 sources are x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x5/10.x, x6/11.x, x6/1.x

<DV REF TEST>

dv-ref is essentially the inverse of dv-src, reporting all of the versions that refer to each file owned by the requested version or range of versions. The referrers are inherently later versions because there are no forward references. dv-ref reports all unreferenced files in one list. Except in the last group version, which may contain new files, there are few unreferenced files unless a project file has been discarded or files not normally part of the group are included in a particular version, which may be done to provide information about the environment in effect at the time of the version capture. testref.py mainly acts on the x group versions created by testput but none of these contain unreferenced files. One test adds non-project files to a version in order to test this condition.

dv.py x0 -ref requests the references to files in version x0. The result should be:
x0/1.x < x1,
x0/2.x < x1,x2,x3,x4,x5,x6,x7,
x0/3.x < x1,x2,x3,x4,x5,x6,x7,
x0/4.x < x1,x2,x3,x4,x5,x6,x7
All versions after x0 refer to its 2.x, 3.x, and 4.x files but only version x1 refers to its 1.x file because that file changes in x2 and later.

dv.py x3 -ref x5 requests the references to files in version x3, x4, and x5. The range is unrelated to the references search space. dv-ref always searches the entire group after each selected version. The result should be:
x3/8.x < x4,x5,x6,x7,
x4/9.x < x5,x6,x7,
x4/1.x < x5,
x5/10.x < x6,x7

dv.py x0 -ref x7 should produce:
x0/1.x < x1,
x0/2.x < x1,x2,x3,x4,x5,x6,x7,
x0/3.x < x1,x2,x3,x4,x5,x6,x7,
x0/4.x < x1,x2,x3,x4,x5,x6,x7,
x1/5.x < x2,x3,x4,x5,x6,x7,
x1/6.x < x2,x3,x4,x5,x6,x7,
x2/7.x < x3,x4,x5,x6,x7,
x2/1.x < x3,
x3/8.x < x4,x5,x6,x7,
x4/9.x < x5,x6,x7,
x4/1.x < x5,
x5/10.x < x6,x7,
x6/11.x < x7,
x6/1.x < x7
This looks for references to all files in x0 through x7 but x7 doesn't contain any of its own files so there is nothing to say about it.

To verify dv-ref listing of unreferenced files, testref creates non-project files z10, z9, z8, z7, z6, z5, and z4 in, respectively, version x0, x1, x2, x3, x4, x5 and x6. Then dv.py x0 -ref x7 produces the same references list as in the preceding test but adds 'Unreferenced: x0/z10, x1/z9, x2/z8, x3/z7, x4/z6,

x5/z5, x6/z4\n'. After performing this test, the non-project files are deleted, allowing other tests and repeated testref to execute without having to execute testput again to restore the original expected conditions.

DV GET TEST

In the get command dv.py VERSION -get OTHER, VERSION is the name of a version in the repository. OTHER specifies a directory not in the repository. This may be an absolute or relative path. If it is just one name it is interpreted as a child of CWD. For example, dv.py x5 -get x6 would copy all of the files in version x5 into CWD/x6, not CWD/bak/x6.

[testget.py] demonstrates and tests the dv.py get operation for proper creation of the destination directory and correctly accessing the delta-based repository. These tests can easily be fully automated for regression testing. It also tests the destination overwrite query action, which normally requires user interaction. That the query is presented and dv.py responds correctly to the user's input can't be automatically tested but testing it one time is sufficient. However, the operations triggered by the response are complex and need regression testing, which is most effective if it doesn't depend on an expected human response. The @ clause was invented specifically to address this issue. It enables test scripts to automatically ensure a specific response and to test the expected consequence.

The result of every dv-get test can be verified immediately, which could allow the same destination directory to be used for all except when it is CWD. However, a different directory is used for each test in order to leave the results for manual inspection, which can help in identifying root cause of errors. Since none of these directories serve any other purpose, they are all created under CWD/gettest to avoid confusion and simplify manually removing all of them. testget.py itself does this when it begins in order to have a known starting condition and to verify that it creates the destination directory without asking if it doesn't already exist.

Although testget.py specifies most destination directories as relative path, it also tests blank OTHER, which defaults to CWD, and absolute path, which it derives from os.getcwd() in order to execute in any directory. All tests are performed against the x versions repository group under CWD/bak created by testput.py. To be sure of the starting condition, testget.py invokes testput.py although this is redundant if testget executes immediately after testput. It doesn't do this again while it executes because it doesn't change the repository.

Most result testing is done simply by comparing files in the destination and source. Some should be identical and others different and some missing. Missing is determined by whether destination is compared to source or source to destination. The compare result is reported as a tuple of three lists, the files that match, those that don't match, and those that are missing. A dv-get error is indicated if this tuple doesn't match the expected pattern. If either directory is missing, all three lists will be empty, which obviously won't match the pattern and will be reported as an error. In most tests, vdict is excluded because it is irrelevant, as dv-get skips it. However, one test includes this in the expected missing list to confirm that dv-get does skip it. Each test comprises the dv-get invocation followed by comparing the version and destination directories. The comparison is performed by testget.py*cmpdirs, which, given two directories, returns a three-list (match, mismatch, missing) tuple to be compared against the expected result. cmpdirs normally skips vdict but takes an optional argument telling to include it. cmpdirs may be called directly but most tests call testget.py*testres, which calls it and

reports whether the tuples are equal and, if not, increments the total error count reported by testget when done.

cmpdirs gets a list of the files in each directory from the Python function os.listdir, which adopts the default OS order, which is may be alphabetical in Windows but not in Linux. list.sort is applied to this ensure consistent order for simple equality test against an expected list. This has nothing to do with version dictionary order, which is by file modification time.

Test dv.py x6 -get gettest 4.x

dv-get is usually invoked to get all of the files that comprise a version, in which case the last argument, EXT, is blank. To get just one file, its name is provided as EXT. This is done to avoid disturbing any of the other project files, typically in the project directory in order to test an older version of the file in the context of the newest versions of others. The first test does this. It first removes CWD/gettest to guarantee that it contains no left-over files or directories in order to simplify the test. It then invokes dv.py x6 -get gettest 4.x. This is a special case, for which cmpdirs and testres are not useful. Instead, the test verifies for itself that CWD/gettest contains only 4.x and that the file matches bak/x0/4.x.

Test dv.py x0 -get gettest/x0

This does not test delta repository access because x0 is the first version created by testput. However, it does have a vdict file, which dv.py deliberately does not include in get because it is not a project file but an artifact of the version control system. The command is dv.py x0 -get gettest/x0. Then gettest/x0 is compared to bak/x0 and bak/x0 to gettest/x0, in both cases with vdict included. In the former case the "missing" list should be empty and in the latter contain only vdict. This verifies that dv-get correctly excludes vdict, which doesn't need to be tested again. The match list should include all four files 1.x, 2.x, 3.x, and 4.x. The mismatch list should be empty. This test also verifies that dv-get correctly makes the CWD-relative directory gettest/x0. This also doesn't need to be tested again but most other tests will inherently test it even though that is not their purpose.

1: Comparing gettest/x0 to bak/x0 (including vdict) should produce (['1.x', '2.x', '3.x', '4.x'], [], []) which says that all files in gettest/x0 match those in bak/x0 and gettest/x0 contains no files that are mismatched or missing in bak/x0.
2: Comparing bak/x0 to gettest/x0 (including vdict) should produce (['1.x', '2.x', '3.x', '4.x'], [], ['vdict']) which says that all project files in bak/x0 are matched in gettest/x0, that bak/x0 contains no files that are mismatched in gettest/x0, and that gettest/x0 is missing vdict.

Test dv.py x1 -get %s/gettest/x1'%cwd

This tests getting a version that has references and coincidentally tests getting to a directory identified by absolute path. cwd is a global variable previously assigned by cwd = os.getcwd(). The directory is under CWD/gettest to allow the test to be executed anywhere but that is irrelevant to dv-get, which does not see it as different from any other absolute path. Although the x1 version comprises six files, its directory contains only 5.x and 6.x. The others are referenced (by vdict) to x0. The results can all be verified for automated regression testing but manual inspection is needed to verify that dv-get correctly shows the source of the files as it copies them into the destination. It should show x0/1.x, x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x.

1: Comparing gettest/x1 to bak/x1 should produce the result (['5.x', '6.x'], [], ['1.x', '2.x', '3.x', '4.x']) which says that 5.x and 6.x match, no files mismatch, and, gettest/x1 contains 1.x, 2.x, 3.x, and 4.x but bak/x1 does not.
2: The inverse, comparing bak/x1 to gettest/x1, should produce (['5.x', '6.x'], [], []) which shows that bak/x1 doesn't contain any files not in gettext/x1 (vdict is excluded).

3: Comparing gettest/x1 to bak/x0 should produce (['1.x', '2.x', '3.x', '4.x'], [], ['5.x', '6.x']) which shows that 1.x, 2.x, 3.x and 4.x match, that no files mismatch, and that x0 is missing 5.x and 6.x.
4: Comparing bak/x0 to gettest/x1 should produce (['1.x', '2.x', '3.x', '4.x'], [], []) which shows the same matching files and no mismatches, and no missing files.

Test dv.py x2 -get gettest/x2

x2 version directory contains the new file 7.x and a unique version of 1.x. It refers to 2.x, 3.x, and 4.x in x0 and to 5.x and 6.x in x1. After get x2 to gettest/x2 see:
1: Comparing gettest/x2 to bak/x2 should produce (['1.x', '7.x'], [], ['2.x', '3.x', '4.x', '5.x', '6.x']) which says that 1.x and 7.x match, no files mismatch, and bak/x2 is missing 2.x, 3.x, 4.x, 5.x, and 6.x.
2: Comparing bak/x2 to gettest/x2 should produce (['1.x', '7.x'], [], []) which says that 1.x and 7.x match and that bak\x2 contains no files that are mismatched or missing in gettest/x2.
3: Comparing gettest/x2 to bak/x0 should produce (['2.x', '3.x', '4.x'], ['1.x'], ['5.x', '6.x', '7.x']) which says that the destination's 2.x, 3.x, and 4.x files match version x0, that 1.x does not match, and that x0 is missing 5.x, 6.x, and 7.x.
4: Comparing bak/x0 to gettest/x2 should produce (['2.x', '3.x', '4.x'], ['1.x'], []) which says that 2.x, 3.x, and 4.x match, that 1.x is mismatch, and that the x0 version contains no files missing in gettest/x2.
5: Comparing gettest/x2 to bak/x1 should produce (['5.x', '6.x'], [], ['1.x', '2.x', '3.x', '4.x', '7.x']) which says that 5.x and 6.x match, there are no mismatches, and bak/x1 is missing 1.x, 2.x, 3.x, 4.x, which gettest/x2 gets from x0 (via the x2 dictionary) and 7.x, which is only in version x2.
6: Comparing bak/x1 to gettest/x2 should produce (['5.x', '6.x'], [], []) which says that 5.x and 6.x match, and bak/x1 contains no files that are mismatched or missing in gettest/x2.

Test dv.py x3 -get gettest/x3

x3 version directory contains only the new file 8.x. It refers to 2.x, 3.x, and 4.x in x0, to 5.x and 6.x in x1, and to 1.x and 7.x in x2. After get x3 to gettest/x3 see:
1: Comparing gettest/x3 to bak/x3 should produce (['8.x'], [], ['1.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x']) which says that 8.x match, no files mismatch, and bak/x3 is missing 1.x, 2.x, 3.x, 4.x, 5.x, 6.x, and 7.x.
2: Comparing bak/x3 to gettest/x3 should produce (['8.x'], [], []) which says that 8.x files match and the version directory contains no files that don't match or are missing in gettest/x3.
3: Comparing gettest/x3 to bak/x0 should produce (['2.x', '3.x', '4.x'], ['1.x'], ['5.x', '6.x', '7.x', '8.x']) which says that 2.x, 3.x, and 4.x match, that 1.x files mismatch, and that gettest/x3 contains 5.x, 6.x, 7.x, and 8.x files, which are missing in version x0.
4: Comparing bak/x0 to gettest/x3 should produce (['2.x', '3.x', '4.x'], ['1.x'], []) which says that 2.x, 3.x, and 4.x match, that 1.x versions are different, and that x0 version contains no files that are missing in gettest/x3.
5: Comparing gettest/x3 to bak/x1 should produce (['5.x', '6.x'], [], ['1.x', '2.x', '3.x', '4.x', '7.x', '8.x']) which says that 5.x and 6.x match, that gettext/x3 contains no files that are mismatched in version x1, and that its 1.x, 2.x, 3.x, 4.x, 7.x and 8.x files are missing in bak/x1.
6: Comparing bak/x1 to gettest/x3 should produce (['5.x', '6.x'], [], []) which says that 5.x and 6.x files match and that bak/x1 contains no files that are mismatched or missing in gettest/x3.
7: Comparing gettest/x3 to bak/x2 should produce (['1.x', '7.x'], [], ['2.x', '3.x', '4.x', '5.x', '6.x', '8.x']) which says that the 1.x and 7.x files match, that no files are mismatched, and that bak/x2 is missing 2.x, 3.x, 4.x, 5.x, 6.x, and 8.x.
8: Comparing bak/x2 to gettest/x3 should produce (['1.x', '7.x'], [], []) which says that the 1.x and 7.x files match, no files are mismatched, and that bak/x2 contains no files missing in gettest/x3.

Test dv.py x6 -get gettest/x6

The delta repository is adequately tested by get x0, x1, x2, and x3 but testget.py also tests get x6, the last unique version created by testput.py, in order to fully exercise the repository. testget skips testing x4 and x5 because they have nothing unique to offer. Version x6 comprises 11 files but bak/x6 contains only the new file 11.x and its unique version of 1.x. Its vdict references 2.x, 3.x, and 4.x in version x0, 5.x and 6.x in version x1, 7.x in x2, 8.x in x3, 9.x in x4, and 10x in x5. As dv-get executes it should show sources as x0/2.x, x0/3.x, x0/4.x, x1/5.x, x1/6.x, x2/7.x, x3/8.x, x4/9.x, x5/10.x, x6/1.x, and x6/11.x. After get x6 to gettest/x6 see:

1: Comparing gettest/x6 to bak/x6 produces (['1.x', '11.x'], [], ['10.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x'])
2: Comparing bak/x6 to gettest/x6 produces (['1.x', '11.x'], [], [])
3: Comparing gettest/x6 to bak/x0 produces (['2.x', '3.x', '4.x'], ['1.x'], ['10.x', '11.x', '5.x', '6.x', '7.x', '8.x', '9.x']) Note that the missing list shows 10.x and 11.x before older files because of the alphabetic order used by cmpdirs. The x6 dictionary order (by file date) is 2.x, 3.x, 4.x, 5.x, 6.x, 7.x, 8.x, 9.x, 10.x, 11.x, 1.x.
4: Comparing bak/x0 to gettest/x6 produces (['2.x', '3.x', '4.x'], ['1.x'], [])
5: Comparing gettest/x6 to bak/x1 produces (['5.x', '6.x'], [], ['1.x', '10.x', '11.x', '2.x', '3.x', '4.x', '7.x', '8.x', '9.x'])
6: Comparing bak/x1 to gettest/x6 produces (['5.x', '6.x'], [], [])
7: Comparing gettest/x6 to bak/x2 produces (['7.x'], ['1.x'], ['10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '8.x', '9.x'])
8: Comparing bak/x2 to gettest/x6 produces (['7.x'], ['1.x'], [])
9: Comparing gettest/x6 to bak/x3 produces (['8.x'], [], ['1.x', '10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '9.x'])
10: Comparing bak/x3 to gettest/x6 produces (['8.x'], [], [])
11: Comparing gettest/x6 to bak/x4 produces (['9.x'], ['1.x'], ['10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x'])
12: Comparing bak/x4 to gettest/x6 produces (['9.x'], ['1.x'], [])
13: Comparing gettest/x6 to bak/x5 produces (['10.x'], [], ['1.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x'])
14: Comparing bak/x5 to gettest/x6 produces (['10.x'], [], [])

Test get to existing directory

The user is normally asked whether to delete existing files in a get destination (OTHER) directory. In order to be fully automatic, testget.py does not confirm the query and response but uses the command line @ clause to test the Y and N response actions.

To test get to existing directory, testget.py creates gettest/x66 directory without involving dv.py. Then dv.py x6 -get gettest/x66 is invoked in order to verify that the pre-clear query does not appear if the existing directory is empty, and the results are the same as getting to a new directory. bak/x6 can't be directly compared to gettest/x66 because it directly contains only 1.x and 11.x. However, the preceding dv.py x6 -get gettest/x6 copies all of the files comprising x6 to gettest/x6 and this can be compared to gettest/x66. Compare gettest/x6 to gettest/x66 produces (['1.x', '10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x'], [], []) .

The next test, dv.py x0 -get gettest/x66 @ N, copies all of the files comprising x0 to gettest/x66 without pre-deleting any of the files copied from x6. dv-get shows getting 1.x, 2.x, 3.x, and 4.x from the requested version x0. Again comparing gettest/x6 to gettest/x66 produces (['10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x'], ['1.x'], []) which is nearly the same as before except that 1.x has moved from match to unmatch as expected because of the four files in x0 only 1.x differs from x6. Comparing

bak/x0 to gettest/x66 produces (['1.x', '2.x', '3.x', '4.x'], [], []) showing that all of its files match those in x66.

After the same command but with Y answer to pre-delete the files, dv.py x0 -get gettest/x66 @ Y, gettest/x66 will contain only the files in version x0. Comparing gettest/x6 to gettest/x66 this time produces (['2.x', '3.x', '4.x'], ['1.x'], ['10.x', '11.x', '5.x', '6.x', '7.x', '8.x', '9.x']) which says that the 2.x, 3.x, and 4.x files, which are the same in all versions, match, that 1.x is mismatched, and that the rest of the x6 files are missing. Comparing bak/x0 to gettest/x66 produces the same result as previously.

Test get default to CWD

If dv-get is invoked with blank destination (OTHER argument) it defaults to CWD and doesn't offer the option of removing existing files, because accidentally answering Y could wipe out project support files. That the user is not asked whether to delete existing files can only be confirmed by observation but that version files are correctly retrieved and other files are not touched can be automatically verified.

In the execution of testput.py, x project files are created in CWD and put into repository versions. Since x6 is the last version, CWD and x6 x files should all match unless something is done to modify the files in CWD, which is what this test does. testget.py optionally initially invokes testput.py. To avoid depending on this, this test initially deletes all x files in CWD and saves a directory list for a simple test that only x files are affected by getting to CWD. Comparing gettest/x6 to CWD produces ([], [], ['1.x', '10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x']) confirming that the x files are now all missing.

dv.py x0 -get shows 1.x, 2.x, 3.x, and 4.x being retrieved from version x0. Comparing gettest/x6 to CWD this time produces (['2.x', '3.x', '4.x'], ['1.x'], ['10.x', '11.x', '5.x', '6.x', '7.x', '8.x', '9.x']) verifying this. 2.x, 3.x, and 4.x are the same in x0 and x6 but 1.x is different and x0 doesn't include any of the others.

dv.py x6 -get shows all of the files being retrieved. Comparing gettest/x6 to CWD produces (['1.x', '10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x'], [], []) verifying that all x6 files have been retrieved.

To verify that throughout all of these changes in CWD only x files are affected, the x files are again deleted and the directory list is compared to the list recorded after deleting x files but before invoking dv-get. To enable repeated execution of testget.py and other test scripts without repeating testget, the .x files in CWD are restored by dv.py x7 .x -get.

DV DELETE TEST

The dv -del operation deletes one version or a range of versions. This is usually used to delete the oldest versions, i.e. from the beginning of the group. However, versions in the middle of the group may also be deleted. With the forward delta model only versions after deleted ones are affected. Project files in the directories of deleted versions are moved to the first undeleted version that refers to them and all dictionaries (vdict file) of undeleted versions that refer to the deleted version file are updated to reflect this src change. The operation is fully tested by examining the directory and vdict of all versions after the deleted range.

Before making any changes, dv-del asks the user to confirm that they want to delete the versions. If the answer is Y, the user is asked if they want to remove the deleted versions' directories. If Y, the user is asked if they want to shift the rest down to fill the gap. A non-blank EXT argument names a destination directory to receive a copy of all unreferenced files in deleted versions. In this case, if the directory already exists and contains files the user is asked whether to delete the files. All of these queries are answered by an @ clause to fully automate regression testing.

The basic dv-del [DvDel] operation without shifting is complicated and testing it is also complicated. Shifting adds considerable complexity to both the operation and testing. To help distinguish between basic operation errors and errors caused by shifting, the operation is tested by two scripts, testdel.py without shifting and testdels.py with shifting. testdel.py not only tests the basic operation but also various command line and interactive responses indicating whether to remove directories and what to do with orphan files. testdel.py performs the same deletions but in all cases removing the directories, shifting the rest, and capturing orphans. Both test multiple deletion examples. Before each example they prepare the repository by invoking testput.py. Having a consistent starting condition simplifies predicting the changes. Each version directory's file list and dictionary (vdict file) provide a core means of testing changes. The initial repository condition, established by testput.py, is (as paraphrased by the arguments passed to testver):

x0 ['1.x', '2.x', '3.x', '4.x', 'vdict']
{"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"]}

x1 ['5.x', '6.x', 'vdict']
{"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"]}

x2 ['1.x', '7.x', 'vdict']
{"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"]}

x3 ['8.x', 'vdict']
{"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"], "8.x": [144201482, "x3"]}

x4 ['1.x', '9.x', 'vdict']
{"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"]}

x5 ['10.x', 'vdict']
{"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"], "10.x": [1534105928, "x5"]}

x6 ['1.x', '11.x', 'vdict']
{"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"],

"9.x": [293824075, "x4"], "10.x": [1534105928, "x5"], "11.x": [1114351625, "x6"], "1.x": [1417620892, "x6"]}

1.x is the only file that exists in different versions. In x0 it contains "1x". Only x1 refers to this. In x2 it contains "1-7-x". x3 refers to this. In x4 it contains "1-9-x". x5 refers to this. In x6 it contains "1-11-x".

<testdel>

Each test in testdel.py calls testdel.py*testrm to verify that version directories are removed or not, depending on the remove query answer. Each version directory in the rest of the group after deleted versions is tested for expected files and dictionary changes by testdel.py*testver.

[testdel.py*DeleteX0]

dv.py x0 -del x0 @ Y N deletes version x0 but does not remove bak/x0, which is confirmed by testrm. x1 refers to all four of x0 files 1.x, 2.x, 3.x, and 4.x. Consequently, these are all added to bak/x1, whose condition (confirmed by testvdir) changes to ['1.x', '2.x', '3.x', '4.x', '5.x', '6.x', 'vdict'] {"1.x": [3646614595, "x1"], "2.x": [4067934080, "x1"], "3.x": [3949760193, "x1"], "4.x": [2754418694, "x1"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"]}

No other versions' directories change but all of them refer to x0 so their dictionaries change.
x2 = ['1.x', '7.x', 'vdict'] {"2.x": [4067934080, "x1"], "3.x": [3949760193, "x1"], "4.x": [2754418694, "x1"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"]}

x3 = ['8.x', 'vdict'] {"2.x": [4067934080, "x1"], "3.x": [3949760193, "x1"], "4.x": [2754418694, "x1"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"], "8.x": [144201482, "x3"]}

x4 = ['1.x', '9.x', 'vdict'] {"2.x": [4067934080, "x1"], "3.x": [3949760193, "x1"], "4.x": [2754418694, "x1"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"]}

x5 = ['10.x', 'vdict'] {"2.x": [4067934080, "x1"], "3.x": [3949760193, "x1"], "4.x": [2754418694, "x1"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"], "10.x": [1534105928, "x5"]}

x6 = ['1.x', '11.x', 'vdict'] {"2.x": [4067934080, "x1"], "3.x": [3949760193, "x1"], "4.x": [2754418694, "x1"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "10.x": [1534105928, "x5"], "11.x": [1114351625, "x6"], "1.x": [1417620892, "x6"]}

[testdel.py*DeleteX2]

dv.py x2 -del @ Y Y N deletes version x2 and removes its directory but does not shift the rest. This should have no affect on x0 and x1. x3 refers to x2 for 1.x and 7.x so these files move into it. All subsequent versions refer to x2/7.x so their dictionaries change slightly but their directories are not changed.
x0 = ['1.x', '2.x', '3.x', '4.x', 'vdict'] {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"]}

x1 = ['5.x', '6.x', 'vdict'] {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"]}

x3 = ['1.x', '7.x', '8.x', 'vdict'] {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x3"], "1.x": [1475035452, "x3"], "8.x": [144201482, "x3"]}

x4 = ['1.x', '9.x', 'vdict'] {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x3"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"]}

x5 = ['10.x', 'vdict'] {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x3"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"], "10.x": [1534105928, "x5"]}

x6 = ['1.x', '11.x', 'vdict'] {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x3"], "8.x": [144201482, "x3"], "9.x": [293824075, "x4"], "10.x": [1534105928, "x5"], "11.x": [1114351625, "x6"], "1.x": [1417620892, "x6"]}

[testdel.py*DeleteX0-X3]

dv.py x0 -del x3 @ Y Y N deletes versions x0 through x3 and removes their directories, which is confirmed by testrm, but does not shift the rest. The only version to refer to x0/1.x is x1, which is being deleted. Consequently x0/1.x is deleted (if EXT were not blank it would be saved). Since x4 refers to the other three x0 files, 2.x, 3.x, and 4.x, these move to x4. x4 also refers to the x1 files 5.x and 6.x, the x2 file 7.x, and the x3 file 8.x. Consequently these also move to x4. Thus, the x4 condition changes to:
['1.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x', 'vdict'] {"2.x": [4067934080, "x4"], "3.x": [3949760193, "x4"], "4.x": [2754418694, "x4"], "5.x": [3174443335, "x4"], "6.x": [2518383236, "x4"], "7.x": [2399161285, "x4"], "8.x": [144201482, "x4"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"]}

Since all of the deleted version files move to x4, x5 and x6 directories don't change but their dictionaries do.
x5 = ['10.x', 'vdict'] {"2.x": [4067934080, "x4"], "3.x": [3949760193, "x4"], "4.x": [2754418694, "x4"], "5.x": [3174443335, "x4"], "6.x": [2518383236, "x4"], "7.x": [2399161285, "x4"], "8.x": [144201482, "x4"], "9.x": [293824075, "x4"], "1.x": [1567976502, "x4"], "10.x": [1534105928, "x5"]}

x6 = ['1.x', '11.x', 'vdict'] {"2.x": [4067934080, "x4"], "3.x": [3949760193, "x4"], "4.x": [2754418694, "x4"], "5.x": [3174443335, "x4"], "6.x": [2518383236, "x4"], "7.x": [2399161285, "x4"], "8.x": [144201482, "x4"], "9.x": [293824075, "x4"], "10.x": [1534105928, "x5"], "11.x": [1114351625, "x6"], "1.x": [1417620892, "x6"]}

[testdel.py*SaveOrphans]

These tests use the command argument EXT "dead" to request that unreferenced files in deleted versions be copied into the CWD/dead directory.

The first test, dv.py x0 -del x6 dead @ Y N, initially removes CWD/dead directory in order to confirm that dv-del creates it. It also makes a dictionary from the vdict in x7 [testdel.py*testvdir]. Since x7 comprises all files in the group but only by reference, its dictionary provides a simple means of

accessing all original files (if the directories of deleted versions are not removed) for testing results. This commands deletes all versions except x7 but doesn't remove their directories. Consequently, x7, which originally contained no project files of its own will contain all of them. This is confirmed by testing it for condition:
['1.x', '10.x', '11.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x', 'vdict'] {"2.x": [4067934080, "x7"], "3.x": [3949760193, "x7"], "4.x": [2754418694, "x7"], "5.x": [3174443335, "x7"], "6.x": [2518383236, "x7"], "7.x": [2399161285, "x7"], "8.x": [144201482, "x7"], "9.x": [293824075, "x7"], "10.x": [1534105928, "x7"], "11.x": [1114351625, "x7"], "1.x": [1417620892, "x7"]}
To confirm that the correct versions of the files have been copied to x7, testdel.py*rptmis is called to compare each file in the captured x7 dictionary to its original source. rptmis reports if the files don't match and if either is missing, which is not actually needed in this case but is in other use-cases. rptmis is called to verify that CWD/dead contains x0-1.x, x2-1.x, and x4-1.x, which are copies of x0/1.x, x2/1.x, and x4/1.x, all of which were referenced by other versions and would have been moved to them except that this command is deleting all of the referencing versions.

The next test, dv.py x0 -del x7 dead @ Y N N Y, deletes all versions in the x group. Existing files in CWD/dead are deleted in order to confirm that this command copies all project files into it because there will be no referrers to any of them. The version directories are not removed so that their files can be compared to the dead copies. After the command executes, the dead directory should contain 'x0-1.x', 'x0-2.x', 'x0-3.x', 'x0-4.x', 'x1-5.x', 'x1-6.x', 'x2-1.x', 'x2-7.x', 'x3-8.x', 'x4-1.x', 'x4-9.x', 'x5-10.x', 'x6-1.x', 'x6-11.x'. The test confirms this and that the files match the originals by comparing each to its source, e.g. 'dead/x0-1.x' to 'bak/x0/1.x'.

The last test, dv.py x0 -del x0 dead @ Y N Y demonstrates and tests saving unreferenced files when a single version is deleted by repeating VERSION as OTHER in order to prevent EXT from being interpreted as OTHER. All project files that testput adds to x group versions are referenced so deleting any one of them does not leave an unreferenced orphan. This test creates the file "noref" in bak/x0. Then after the command executes this file should appear in CWD/dead as x0-noref.

<testdels>

[testdels.py] does the same three deletions, x0, x2, and x0-3, as testdel.py but with no user response variations because these are adequately tested by testdel.py and the basic testing performed by testdels.py is more complicated. Each dv-del invocation command line includes an @ Y clause to answer Y to all four questions, delete versions, remove directories, shift the rest down to fill the gap, and delete existing files in dead (the orphan directory).

As with testdel.py, each test begins by invoking testput.py (through testdels.py*prepx) for a consistent initial repository (x0-x6). Because each test includes orphan capture, testdels.py*norefs is called to add the "noref" file to each version being deleted. This has the same name in all but it contains text that indicates its origin, e.g. x0noref. This can be used to verify that dv-del correctly names the orphans, e.g. x0-noref. This is needed because testput.py itself doesn't create any unreferenced files and some deletions don't produce orphans although some do. For example 1.x in x0 is referenced only by x1. If x0 is deleted, x1 inherits its 1.x so it is not orphaned, but it is orphaned if both x0 and x1 are deleted. The 1.x in x2 is referenced only by x3 and deleting both of these versions orphans x2-1.x.

testdels.py duplicates the testing of remaining versions' directory and dictionary content done by testdel.py, but the version names are shifted and this shows up in both their directory names and sources listed in their directories. The expected patterns must be created theoretically but testdel.py can

provide additional confirmation by comparing inherited files to their originals (in deleted versions whose directories have not been removed). Obviously, testdels.py can't do this but it has another means of confirmation. Before doing the deletion, each test calls testdels.py*preget to invoke dv-get on each version after the deleted range with the destination directory dspre/ver. For example, initially x1 contains its own 5.x and 6.x but refers to x0 for 1.x through 4.x, and dspre/x1 will contain all six of these. dv-get only gets project files, skipping vdict and the added noref. After deletion, testdels.py*postget is invoked to verify that the content of shifted versions has not changed. It takes two arguments, vers is a list of new version numbers, and integer shift is the offset required to compute the original name. For example, if x0-x3 are to be deleted, preget is called with vers [4,5,6] and it will create x4, x5, and x6 in dspre. After deletion, postget is called with vers [0,1,2] and shift 4, telling it to test the new x0, x1, and x2 against the original x4, x5, and x6. postget could get each new version and compare its files to its dspre original. Either may be missing files relative to the other and the files that both have may or may not be identical. If we only wanted to report whether there has been any change this would not be a very complex process but diagnosing a problem could be very difficult. Instead, postget invokes dv-cmp to compare each new version to its dspre counterpart, which is an ordinary directory that happens to have a version name. dv-cmp displays a detailed report on the comparison, but provides no programmatic indication of the status. Consequently, postget invokes dv with its output redirected into a file, which is then parsed to determine whether contents match. This is slower than having postget do the comparison itself but less complex than duplicating the status reporting of dv-cmp. This test is especially effective for regression testing because it does not depend on theoretical analysis but actual results.

[testdels.py*DeleteX0]

dv.py x0 -del x0 dead @ Y deletes x0 and shifts x1-x6 to x0-x5.

The new x0 should be ['1.x', '2.x', '3.x', '4.x', '5.x', '6.x', 'vdict'], {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"]}
retaining the original x1 5.x and 6.x and inheriting the original x0 1.x-4.x. That the dictionary still shows 1.x-4.x sourced by x0 superficially looks like a failure to move them but is the correct combination of moving into the original x1, which shifts to become x0.

The new x1 should be ['1.x', '7.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x1"], "1.x": [1475035452, "x1"]}

The new x2 should be ['8.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x1"], "1.x": [1475035452, "x1"], "8.x": [144201482, "x2"]}

The new x3 should be ['1.x', '9.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x1"], "8.x": [144201482, "x2"], "9.x": [293824075, "x3"], "1.x": [1567976502, "x3"]}

The new x4 should be ['10.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x1"], "8.x": [144201482, "x2"], "9.x": [293824075, "x3"], "1.x": [1567976502, "x3"], "10.x": [1534105928, "x4"]}

The new x5 should be ['1.x', '11.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x1"], "8.x": [144201482, "x2"], "9.x": [293824075, "x3"], "10.x": [1534105928, "x4"], "11.x": [1114351625, "x5"], "1.x": [1417620892, "x5"]}

dead should contain file x-noref, which contains the text x0noref. x0-1.x is not orphaned because the old/new x1/x0 inherits it.

[testdels.py*DeleteX2]

dv.py x2 -del x2 dead @ Y deletes x2 and shifts x3-x6 down to x2-x5.

x0 is not changed and should still be ['1.x', '2.x', '3.x', '4.x', 'vdict'], {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"]}'

x1 is not changed and should still be ['5.x', '6.x', 'vdict'], {"1.x": [3646614595, "x0"], "2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"]}

The new x2 (originally x3) should be ['1.x', '7.x', '8.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "1.x": [1475035452, "x2"], "8.x": [144201482, "x2"]}

The new x3 should be ['1.x', '9.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x2"], "9.x": [293824075, "x3"], "1.x": [1567976502, "x3"]}

The new x4 should be ['10.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x2"], "9.x": [293824075, "x3"], "1.x": [1567976502, "x3"], "10.x": [1534105928, "x4"]}

The new x5 should be ['1.x', '11.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x1"], "6.x": [2518383236, "x1"], "7.x": [2399161285, "x2"], "8.x": [144201482, "x2"], "9.x": [293824075, "x3"], "10.x": [1534105928, "x4"], "11.x": [1114351625, "x5"], "1.x": [1417620892, "x5"]}

dead should contain file x2-noref, which contains the text x2noref. The noref files in x0, x1, and x3 are not orphaned because these versions are not deleted.

[testdels.py*DeleteX0-X3]

dv.py x0 -del x3 dead @ Y deletes x0-x3 and shifts x4-x6 down to x0-x2

The new x0 should be ['1.x', '2.x', '3.x', '4.x', '5.x', '6.x', '7.x', '8.x', '9.x', 'vdict'],
'{"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x0"], "8.x": [144201482, "x0"], "9.x": [293824075, "x0"], "1.x": [1567976502, "x0"]}
inheriting the original x0 2.x-4.x, x1 5.x and 6.x, x2 7.x, x3 8.x, and retaining the original x4 9.x and 1.x.

The new x1 should be ['10.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x0"], "8.x": [144201482, "x0"], "9.x": [293824075, "x0"], "1.x": [1567976502, "x0"], "10.x": [1534105928, "x1"]}

The new x2 should be ['1.x', '11.x', 'vdict'], {"2.x": [4067934080, "x0"], "3.x": [3949760193, "x0"], "4.x": [2754418694, "x0"], "5.x": [3174443335, "x0"], "6.x": [2518383236, "x0"], "7.x": [2399161285, "x0"], "8.x": [144201482, "x0"], "9.x": [293824075, "x0"], "10.x": [1534105928, "x1"], "11.x": [1114351625, "x2"], "1.x": [1417620892, "x2"]}

dead should contain x0-1.x (1x), x0-noref (x0noref), x1-noref (x1noref), x2-1.x (1-7-x), x2-noref (x2noref), and x3-noref (x3noref).

To top of [TESTING]

PROGRAM DESIGN

[Topic End] [Topics]

<SYSTEM>

bv.py, dv.py, and the scripts that test them are designed to work in Windows and Linux. Some versions of Linux default to Python 2 even though they also support version 3, for which these scripts are designed. To correct this, the first line of every file is:
#!/usr/bin/env python3
This is only interpreted by Linux. Windows and the Python interpreter ignore it. To be correctly interpreted it must have a Unix line ending. This can only be achieved if the entire file has this although the Python interpreter is agnostic and these scripts have been created in Emacs, which is also agnostic.

The scripts mainly comprise standard Python code and functions, most of which work the same way in Linux and Windows. One notable exception is the os.listdir function, which the scripts use extensively. The order of the list is determined by the operating system. It is alphabetical in Windows but random in Linux. This doesn't matter in bv.py and dv.py, which sort file lists by file modification times. Test scripts extensively test commands by comparing the list of files in a directory to the expected list. Even if these lists were not in the same order they could be parsed to determine whether they contained the same files. However, it is cheaper to apply the sort method to the directory list to make it consistent and then simply test the two lists for equality (==).

Although most of the code is standard Python with consistent behavior, there are situations where the operating system is invoked either because there is no alternative or because it is much cheaper than the pure Python alternative. All of the scripts begin by executing oswin = sys.platform.lower()[0:3] == 'win', making the global boolean oswin available wherever a Windows or Linux alternative must be chosen.

bv.py and dv.py care about Windows vs. Linux only in the detcmp function (bv.py dv.py) which is called to invoke another program for a detailed interactive comparison of two versions of a file (bv/dv VERSION -cmp OTHER file). The program is hard-wired as winmerge in Windows and meld in Linux. These particular choices may be changed but for anyone working in both operating systems it is helpful that detcmp chooses one or the other based on oswin.

The test scripts have many instances where they need to choose between Windows and Linux alternatives. In particular, bv.py and dv.py are invoked assuming that they reside in the same directory as the test script and that they are not in PATH, which are reasonable assumptions for programs under development (and testing before full deployment or as tutorial). In Windows, the Python statement os.system(bv or dv...) invokes bv.py or dv.py in the same directory by default. In Linux, the path is required as well as the file's extension. This is handled, for example, by the execmd in testbv.py, testcmp.py, testget.py, and testdel.py. This function invokes any command involving a local script, for example "dv.py x6 -get gettest 4.x", with the statement os.system(cmd if oswin else './%s'%cmd).

The test scripts contain instances where sets of files are identified by globbing, e.g. "*.x". Python's glob module is not always available and does not always work correctly. Intrinsic Python code can parse file names to provide the equivalent selection means but it is cheaper to invoke the operating system to do this as part of the operation to be applied to them. Windows and Linux similarly interpret most glob expressions but they don't always have the same command name. For example, files are deleted by del in Windows but rm in Linux. In several instances, scripts want to delete all of the .x files in the project directory. They do this with os.system('%s *.x'%('del' if oswin else 'rm').

<MOST RECENT VERSION>

[bv.py*MostRecentVersion] [dv.py*MostRecentVersion]

For both bv.py and dv.py, a VERSION argument ending in "+" indicates an incremental version whose actual name is synthesized by incrementing the suffix of the most recent version of the argument's base name. If the argument is simply "+", the most recent version directory, regardless of its name, is taken as the base. Usually the argument identifies a particular sub-project group. bv.py needs to determine the most recent version only if VERSION is incremental but dv.py needs to know it for many operations. Most notably, dv-put always needs to read the most recent version's dictionary in order to determine, for each file, if the repository already contains an identical file, in which case the file is not copied into the new version's directory but, instead, a reference to the existing file is recorded in the new version's dictionary. For dv.py get and cmp, an incremental VERSION argument is a convenience feature to select the most recent version of the group.

The core of the MostRecentVersion process is identical in bv.py and dv.py. However, before it is engaged, VERSION is inspected to determine whether it is incremental. If not, bc.py skips the process but dv.py sets up conditions for finding the most recent version. VERSION is assigned directly to the verbase search string unless it has a numeric suffix, which is stripped off of the name assigned to verbase.

The most recent version is found by comparing the modification time (os.stat(vdir)[stat.ST_MTIME]) of every version directory whose name passes the base name filter. In most cases this selects the same version as would be selected by looking for the one with the greatest numeric suffix but it is cheaper (involving no parsing of the name) and more flexible. For example, if the user creates the first version of a group without a numeric suffix, it will nevertheless be selected as the most recent. If the user modifies an existing version in the middle of a group it becomes the most recent. This is an unusual action and whether the user expects the next incremental VERSION to refer to the modified version or the numerically last one cannot be determined. They will be asked whether to overwrite an existing version but not whether to create a new incremental one. The former is unusual and the query is appropriate. The latter is very common and should not be interrupted. Consequently, the former is the better default for "most recent version" after a mid-group version modification.

To find the "most recent version" all directories in the repository whose name passes the base name filter are examined. The one with the newest modification time is the "most recent". The file system time stamp has rather low resolution, which is not a problem when directories and files are created by the user. However, automated creation, notably by scripts that test bv.py and bv.py, can produce multiple directories with the same time stamp. In most cases, this anomaly can be resolved by primarily sorting by time and secondarily alphabetically. This is done by initially applying the sort method (alphabetically by default) to the repository version list and choosing "most recent" by time stamp >= the previous "most recent". Thus, if two directories have the same time, the later one alphabetically is taken as the most recent.

DV DESIGN

<DELTA MODEL>

There are two models of de-duplication. In the forward delta model, each successive version contains only files that are different from earlier versions and references to earlier versions for files that are not different. In the reverse delta model, all specified files are copied into the tip version directory and earlier duplicates are replaced by references. Each model has its own advantages and disadvantages.

One reverse delta advantage is having all files of the most recently captured version immediately available for comparison to the current working versions using any program or system function. This can be convenient but a forward delta version control program can provide the same capability by getting a version to a temporary directory. Reverse delta also simplifies pruning old versions from the repository. The oldest directories can simply be removed. With forward delta, all later versions have to be examined and updated if they refer to a version being deleted. This is not a significant issue because the main motivation for pruning is to recover storage space, which is significant only in a non-delta system.

The main advantage of the forward delta model is that putting the next version can be much cheaper. If each version is stored with a dictionary that tells the source of all of its files, putting the next version requires examining only this dictionary to determine whether any of the files that comprise the version are duplicates of older ones and, therefore don't have to be replicated. With the reverse delta model, all previous versions must be examined for files (or forward references to files) identical to any that comprise the new version. With forward delta, overwriting the most recent version, which is done relatively frequently to undo a recent change, can be done by the normal put process. With reverse delta, the standard put would have to be augmented with a search for previous references to the version being overwritten.

Pruning a forward delta repository and putting a new version in a reverse delta repository are equally complex. Putting a new version is done much more often than pruning the repository. Therefore, dv.py implements a forward delta repository.

<VERSION DICTIONARY>

Regardless of the chosen delta model, files have to be repeatedly compared to avoid duplication. Byte-by-byte comparison is infallible but very expensive. Files can also be compared by calculating and comparing their CRCs. This is not absolutely infallible but good enough and much cheaper because the CRC of a file only needs to be calculated once and saved for future comparisons. Python's zlib module (which provides access to InfoZip zlib) function crc32 calculates a 32-bit unsigned long integer. The

large range of this number affords good (collision-free) reliability but comparing two of these intrinsic integers is very cheap.

Since each version directory stores only files that are unique, a list of the files that comprise it but are stored in other versions must be recorded. With a reverse delta repository the tip always stores all selected files and this list is not needed but it must be added to earlier versions that contain duplicate files, which need to be deleted. With a forward delta repository, the list of referenced files needs to be created when the version is put. With either model, finding a duplicate source in the first place requires searching but, if the source is recorded, this search need not be repeated.

It costs little to record the CRC of all files and the source of referenced ones, especially with the forward delta model because this recording is needed only when a version is put. It can change when earlier versions are deleted but this is the least frequent operation. It would be feasible to record all files that comprise a version as specially formatted strings, e.g. "file>CRC<source" but this would have to be parsed in use. The most common uses are to get a list of the files that comprise a version and to get the CRC and source of a particular file. The most efficient means of supporting these uses is a dictionary in which each key is the file name and the value a list comprising the CRC as a long integer (even if stored in the record as a string) and the (version) name of the source. The file that records this information need not store it as a Pythonic dictionary because any format could be translated into a dictionary when it is read. However, the json module can write and read an intrinsic dictionary to and from a text file. Using this to read the file into a dictionary object [dv.py*verdic] simplifies dv.py and the (likely compiled) json.load function is probably faster than the interpreted Python equivalent. The dictionary is created from separate non-dictionary data. This can only be done by a Python process [dv.py*Put] whose complexity and execution time are not reduced by creating a dictionary for json.dump to write to the file. However, to gain the benefits of using json.load to read the file into a dictionary object without embedding json format details in the code, it is best to do this.

The dictionary is essential for files that the version references but less important for those that it stores itself, which can always be found by listdir of the version directory. However, the CRC of each file is calculated when the version is put and, to avoid repeating this effort, the value must be recorded. Some operations, e.g. showing the source of all files that comprise a version, are simplified if the dictionary records direct files in the same way as referenced ones, including not only CRC but also source although this obviously is the version itself. An ancillary benefit of this design is that the dictionary can be ordered strictly by file date, which isn't essential to its operation but provides an intuitive overview of project development.

<prepgrp>

Obviously, for the put operation when the VERSION argument is incremental (+ suffix) the most recent version of the group encompassing VERSION must be determined. For the benefit of incremental put, dv.py*MostRecentVersion assigns the next version name and directory to vname and vdir. However, put, ref, and del all need to know the most recent version so its name and directory are assigned to mrver and mrdir. ref and del need to examine the entire group encompassing VERSION. The MostRecentVersion process iterates over this group and could simultaneously build an intrinsic description of it, but this would be a burden on the process, which is optimized for finding the one most recent version. Instead, this is left to dv.py*prepgrp, which ref and del invoke as needed. Based on vname (VERSION or assigned by MostRecentVersion) prepgrp builds global grplist, a list of all of the version names in the group. It assigns to global integer grpbeg the index of vname in this list. In the ref and del command line, OTHER may specify the (inclusive) end of range of selected versions. It may be

a complete version name or just its numeric suffix, e.g. "dv10" or "10". OTHER has been assigned to global oparg. If oparg is just a number, prepgrp reassigns it the vname root plus number. If it is not blank, prepgrp assigns to global integer grpend the index of this version in grplist + 1, to make it the exclusive range end compatible with Python's slice model, i.e. grplist[grpbeg:grpend] is the sub-list of selected versions.

<DV PUT>

All operations other than put require the version directory indicated by VERSION to exist. In most put cases it doesn't exist and dv.py*Put silently creates it. Otherwise, it asks the user if they want to overwrite it and, if yes and it contains files, whether they want to delete those files. dv.py*getans is called to get the interactive or programmatic (@ clause [Testing]) answer.

Put is the only operation whose command line includes the FILES argument [CommandLine]. If the argument is blank then all files in project directory are put after asking the user whether this is their intent. If it is "*" then all files are put without asking. The FILES argument can be any number of separate file selectors, which may be a specific file or an extension filter ("*.ext*" or "." for no extension) These are all collected into the fsel list. Put calls dv.py*filesByDate to get a list of the files in the project directory (CWD) sorted by date, which it assigns to flist. It then iterates over the fsel list and, for each filter, searches flist for files that pass the filter, adding those that do to the src list, which tells the names of all files that comprise the version. This is is primarily ordered by the order of FILES filters and secondarily by the modification dates of files that pass the filters. This order will carry over into the dictionary. The dictionary is mainly accessed by file name key, for which its order is irrelevant. However, it is sometimes accessed by order, primarily to interpret it for the user (e.g. src and ref operations). In this case, the purpose of the order is to provide an intuitive pattern for the user. File date is clearly useful for this. Dividing the dictionary into file groups can also be useful but not more than ordering strictly by date. Because the two schemes are equally useful and ordering strictly by date would require a complex sorting process, the file type/date order is used. However, if FILES is blank, *, or a single file type, the dictionary will be ordered strictly by date. If no files in the directory pass any of the filters, dv.py reports this to the user and exits.

The src list contains the names of all project files to be put into the version. How this is done is determined by comparing to the most recent version. dv-put calls dv.py*verdic to make a dictionary object (mrdic) from the most recent version's vdict file. It then iterates over src, using each file name as index into mrdic. If the file is new, it won't be in mrdic. The lookup is done in a try block because it is cheaper to catch the KeyError exception than to do the lookup twice (in followed by get or index). If the file is in mrdic then its recorded CRC (value[0]) is compared to the CRC calculated for the file being put. If they are equal, the file is added to vdict, the dictionary being made for the new version, with the source version (value[1]) from mrdic as the source in vdict. This may be the most recent version itself or any earlier version. An advantage of forward delta is that versions are stable and there is no need to look back further than the most recent one. If the file is not in mrdic or it is but the CRCs don't match, it is added to vdict with the new version as source. It will be copied into the new version directory but this is not done immediately because the directory is not created until it has been determined that the new version would not simply duplicate the most recent. Instead the file name is added to the newfiles list. After going through all files in src, if newfiles is empty the user is asked whether they want to make this new version even though all of its files are identical to the most recent version. If they answer No, dv.py exits, leaving the repository unchanged. If newfiles is not empty the new version is made without question.

Normally, the new version is actually new and its directory is created. The directory may already exist because an existing version is being overwritten or is left over from some other operation. dv-put ignores this difference when it makes the new version. The files in newfiles list are copied from the project directory into the version directory using shutil.copy2, which not only copies the contents but also statistics of the file. It is especially important that the copy retain the modified date of the original instead of adopting the date of the put operation. json.dump is invoked to write vdict, the new dictionary object, to the version's vdict file. If overwriting an existing version, all files it contains (the user declined to delete them) that have the same name as the new version will be replaced. Any unique information in its dictionary is lost and any other files not being replaced will remain as orphans not recognized as part of the version. Sometimes this is desirable because the directory may contain non-version files for reference.

<SRC REF GET CMP DEL>

In the canonical dv.py command line [CommandLine] VERSION FILES OPERATION OTHER EXT, VERSION is the only argument shared by all operations. Only put supports FILES and only src, ref, get, and del support OPERATION, OTHER, and EXT. If OPERATION specifies one of these (blank means put) the command line needs further processing that they can share. It is more efficient to do this inline for all of them instead of in functions that they could individually invoke.

Only OPERATION is always required (unless put). OTHER is needed for most operations but EXT is entirely optional. It can exist only if OTHER is not blank. This can be determined by the argument count, which is len(sys.argv) unless an @ clause is appended to the normal command line to facilitate regression testing [Testing]. To normalize the argument list, dv.py*CommandLine assigns maxarg the (argv) index of the last normal argument, allowing other command analyzers to ignore the possibility of an @ clause.

If maxarg > 3, sys.argv[4] is assigned to extarg. dv.py*existerr is called to report and exit if the VERSION directory assigned to vdir by dv.py*MostRecentVersion does not exist because all operations other than put can only operate on an existing version. In most cases these operations don't access the version's dictionary as a dictionary but as a list of src/file strings. dv.py*verlist is called to make this list, which is assigned to flist for all of the operations to use in their own way.

The OTHER argument has been assigned to oparg. For src, ref, get, and cmp, if it is blank or ".", odir is assigned CWD. Otherwise, if op is not GET (which requires OTHER to be a non-version directory) and the version indicated by OTHER exists, its repository directory is assigned to odir. Otherwise, odir is assigned oparg, which will be treated as a non-version directory. odir is not assigned if op is DEL because dv-del interprets OTHER uniquely.

SRC

[dv.py*Src] The src operation essentially displays the version's dictionary in a friendly way. The flist format, with each item as src/file, is ideal for this. src simply displays it as a comma-delimited string.

REF

[dv.py*Ref] The ref operation appears to be fairly simple, just telling the versions that refer to each file in one or more versions, but is programmatically relatively complex. The dictionary of each version later than VERSION in the group of which VERSION is a part is examined for references. This process inherently requires simultaneously iterating over the source dictionary, the group list, and the dictionary

of each version in the group. VERSION always identifies a source version. If OTHER is not blank then all versions in the group from VERSION to OTHER are sources.

For each file in each version between VERSION and OTHER, all later dictionaries could be examined for a reference. Although conceptually simple this would be the most expensive process because the potential referrer dictionaries would have to be loaded repeatedly or initially loaded into a very large list. A cheaper approach is to build a temporary dictionary (refdic) whose keys are the source *version*/*file* and value a list of referring versions. This enables the process to be turned inside-out, iterating over referrers instead of sources.

[prepgrp] is called to make the group list. Although no versions before VERSION will be examined, prepgrp includes the entire group in grplist because it serves other operations that do need this information. It assigns to grpbeg the index of the grplist element corresponding to VERSION and to grpend the index of OTHER + 1 (to be compatible with slice). The potential referrers are all versions in grplist from grpbeg to the last (slice grplist[grpbeg+1:]). The dictionary of each one of these is loaded and all items examined for source between VERSION and OTHER (which is only VERSION if OTHER is blank) but not a self reference. Each of these references is added to refdic. If refdic already contains the corresponding *version*/*file* (key) this version is added to its value list. Otherwise, the complete item is added to refdic.

After iterating over potential referrers, the key of each item in refdic is the *version/file* of each version between VERSION and OTHER and its value a list of referrers. Each item is displayed as "*key* < *referrers*". The ref operation also displays in one list all files in the VERSION through OTHER range that are not referenced. The process to create refdic provides no direct help for this but refdic itself is helpful. A list of all unreferenced files is built by iterating over all files in the directory of each version between VERSION and OTHER and adding to the temporary list unref, the *version*/*file* of each that does not appear in refdic. A valuable aspect of refdic as a dictionary is that looking up *version*/*file* is cheap because it is a hashed key. If unref is not empty, it is displayed as a comma-delimited list (of "Unreferenced").

GET

[dv.py*Get] Get is a simple operation. It doesn't change the repository in any way and only involves one version (VERSION). Even if OTHER coincidentally names a version, it is interpreted as a project sub-directory. Unless the destination is the project directory (OTHER is blank or ".") dv.py*mkclrdir is called to make the new destination directory or, after asking the user, clearing an existing directory. Usually, all files in the version specified by VERSION (and already added to flist) are copied into the directory. However, if extarg is not blank, an EXT argument names a specific file and only this is copied to the destination. This is usually used to restore one file in the project directory to an older version.

<DV CMP>

[dv.py*Cmp] Although dv-cmp only shows existing conditions and doesn't change the repository, it is programmatically complex. It involves two versions or one version and a non-version directory and it groups files by their relationships, which is very informative for the user but demands significant complexity.

Initially, dv.py*verlist is called to make a version/file list from the OTHER dictionary (odir dict file). This is assigned to oplist and it is None if OTHER specifies a non-version directory. The distinction

between OTHER as version or non-version directory is important to how the cmp operation proceeds. If the EXT argument is blank, all files in VERSION and OTHER are compared and their relationship reported without much detail. Otherwise, dv.py*detcmp is called to open a program to provide a detailed, and probably interactive, comparison of two versions of the one file named by the EXT argument. In this case, dv-cmp does nothing other than verifying that the file exists in both VERSION and OTHER and then calling detcmp.

When dv-cmp compares all of the files, it sorts the pairs by their relationship in order to present to the user an overview of the general relationship between the two versions. In all cases, the files are added to match, mismatch, or missing lists. If OTHER is a version there is also a "same" list of files that both reference to the same version (source). The source version may be different from both or one of the two that the other refers to.

The relationship lists are filled by iterating over flist, the list of files (each *source/file*) in VERSION dictionary, and comparing each file to those listed in oplist if OTHER is a version or to the files in the OTHER directory (if not a version). oplist items are *source/file*. If the flist item is in oplist, the *source/file* is added to the "same" list.

If oplist is None or doesn't contain the flist item, a complex analysis is required to determine whether to add the flist file to match, mismatch, or missing list. In all cases, the flist (VERSION) file is repdir + flist item, e.g. bak/x6/1.x. The OTHER file is found by parsing the file name from the flist item using os.path.basename, which works because *source/file* looks like a path. If this file doesn't exist the file name is added to the "missing" list. Otherwise, filecmp.cmp is invoked to compare the two files. If this returns True, the file name is added to the "match" list.

If the files don't match the pair is added to the mismatch list, which is more complicated than missing and match list items, which are simply *source/file*. The rationale for this is that if OTHER doesn't include the file or it includes a matching file there is no reason to look deeper into the relationship but if it contains a different version of the file, an investigation may be warranted. To provide the user with a quick overview of the relationship, each mismatch item tells whether the VERSION file is newer or older than the OTHER file and indicates each file's source. If the version name appears at the beginning of the *source/file* string, the version is *source* and the file is presented as *source/file*. Otherwise, it is presented as *version>source/file*. For example, if x5 refers to x4/1.x and x7 to x6/1.x, dv x7 -cmp x5 will add the mismatch item "x7>x6/1.x newer than x5>x4/1.x" while dv x7 -cmp x4 will add "x7>x6/1.x newer than x4/1.x".

After filling the relationship lists, the ones that are not empty are displayed. Same, Match, and Missing items are rather simple and displaying the lists as single lines (with line wrap only if longer than display width) naturally provides a good overview. This is also the case with Mismatch if it contains only one item. However, the complexity of each item makes a confusing single-line display of multiple items so each item is a single indented line under the one "Mismatch:" title.

The final relationship displayed is the list of files in OTHER that are missing in VERSION. Obviously, this list would not be built iterating over flist. First a list of OTHER files is made. If OTHER is a version, this is made from the oplist *source/file* list made from its dictionary by removing each item's "*source/*" prefix, leaving just the file name: oflist = [f[f.find('/')+1:] for f in oplist]. If OTHER is a non-version directory, oflist is made from all of its files. oflist.sort is invoked mainly to guarantee a consistent list for regression testing but it also can reduce the possibility of the user attributing some significance to irrelevant order differences. The Missing list is made by removing all of the files in

oflist that are also in flist. For each item, which is just file name, in oflist, dv.py*verfile is called to search flist, whose items are *source*/*file*, for just the file regardless of source. It is especially important that this be at the end of the display if OTHER is a non-version directory. Most version-directories contain few non-project files, but non-version directories may contain many of these and the resulting long list would disrupt the relatively concise comparison provided by the other groups.

<DV DEL>

[dv.py*DEL] Deleting versions is the most expensive operation in both programmatic complexity and execution time. These could be significantly reduced by a reverse rather than forward delta repository but this would move the complexity to the put operation, which is much more frequent. With either delta model, there is little repository memory waste and not much impetus to delete versions no matter how old they are.

Deleting versions does not lose their information. All of their files referenced by later versions that are not being deleted are copied to the earliest (undeleted) referrer and their unreferenced files are optionally copied into the directory specified by EXT argument. However, deleting versions is an irreversible operation, which should not be done without confirming the user's intent. If OTHER is blank, the command line is only asking for VERSION to be deleted and this is indicated in the user query. However, OTHER may be the same as VERSION in order to delimit an EXT argument, in which case the query only indicates the single version. Otherwise, if OTHER is not blank the user is asked whether they want to delete the range from VERSION to OTHER.

Initially, dv-del calls dv.py*getans to make sure that the user intends to delete versions. If not, it immediately exits. It then asks the user if they want to remove the deleted versions' directories. If Y then it asks if they want to shift the rest down to fill the gap. This question needs to be answered before deletion, which includes updating the remaining versions' dictionaries to replace deleted sources and simultaneously including shifted names in this analysis is more efficient than doing it in a separate process. Directories are not removed until after versions are deleted but this question is asked beforehand because the shift question is asked only if the directories are to be deleted.

[prepgrp] is called to build grplist from the base name in VERSION and assign grpbeg and grpend (exclusive like slice) the grplist indices of the deleted range. All versions < grpbeg are irrelevant because there are no forward references. All grpend and later are not deleted and may refer to deleted versions. dv-del does this before asking the shift question because it combines the boolean question of shift with the actual shift value by assigning shift 0 if the user declines or grpbeg - grpend otherwise. For example, if x1 through x3 are deleted from the group x0, x1, x2, x3, x4, x5, x6, grpbeg is 1 and grpend is 4 and shift is assigned -3. Then the new names for x4, x5, and x6 will be grplist[4-3] = x1, grplist[5-3] = x2 and grplist[6-3] = x3.

Only versions after the deleted ones need to be changed. These are grplist[grpend] to the end. The process iterates over these (with index idx) assigning each to vname to simplify subsequent code. If shift is not 0, the version's new name, which is grplist[idx+shift] is assigned to repl. For each, the dic dictionary is made from the version's vdict file [dv.py*verdic]. In nearly all cases, the dictionary will be changed and need to be written back out to vdict but if shift is 0 and a version contains no references to earlier versions then its dictionary is not changed. This is indicated by dchng flag initially assigned False and reassigned True if there are any changes.

The dictionary is processed by iterating over it using for fname,dval in dic.items(). Each dictionary item is *file*:[*crc*,*src*]. Since the value is a list, dval is an lval and any changes to it change dic. The key is a string and changing fname would not change dic but this is irrelevant because the file name will not be changed. If *src* (dval[1]) is a deleted version or an undeleted one whose name is changed by shifting then it is replaced by the version that inherits the file (fname) or the new name. There is no deterministic means of determining this replacement. It is instead done with the dictionary repdic, which is built as the version dictionaries are processed. The key is *src*/*file*, e.g. "x0/1.x" refers to the 1.x file in version x0. The value is the new file owner. For each file in a version's dictionary, if *src* is vname (this version) nothing is done if shift is 0; otherwise, dval[1] is assigned repl and repdic['vname/fname'] = repl is added to repdic to guide any later versions that refer to this file. If src is not vname then if it is not in grplist, bad reference is reported. Otherwise, its grplist index is used to determine its status. If it is not in the deleted or shifted range then nothing is done. Otherwise, the value of repdic['src/fname'] is the new source. It this key doesn't exist, vname is the earliest version to refer to fname in the deleted src. It would not be a reference to a shifted version because versions are examined in order and references can only be to earlier versions. As the earliest version to refer to this file vname inherits it. The file is copied into vname's directory and repdic['src/*fname*'] is assigned vname (if shift is 0) or repl.

This updating process is guided entirely by references in dictionaries of undeleted versions. It does nothing to identify unreferenced files (orphans) in deleted versions, which are supposed to be copied into the dead-end directory specified by EXT (if EXT is not blank). Instead, it makes the complementary movlist array of files that have been moved from deleted to undeleted versions. Each element of movlist is a list of the files moved from a deleted version. This is initially defined by movlist = [[] for idx in range(grpend-grpbeg)]. movlist[0] is the list of files moved from the first deleted version, i.e. grplist[grpbeg]. When a file (fname) is moved, its src grplist index is srcidx, so movlist[srcidx-grpbeg].append(fname) adds it to the deleted version's file list. After the undeleted versions have been processed, files in each deleted version directory that are not in movlist are orphans. If EXT is not blank dv.py*mkclrdir is called to either make the new orphan directory specified by EXT or ask whether to delete the existing directory's files. This is where the final user query related to delete is issued (through getans so that an @ clause can provide an automatic answer). Then dv-del iterates over movlist and files in each version directory that are not in its move list are copied into the orphan directory under a new name composed by prefixing the source. For example, 1.x file in deleted version x0 would be renamed x0-1.x. This not only provides useful origin information but also affords a simple means of saving orphans with the same name from different deleted versions.

If deleted version directory removal has been requested, the directories of the versions in grplist from grpbeg to grpend (exclusive) are removed. This could be done while processing undeleted versions if saving orphans has not been requested but it is simpler to wait until the end regardless. After removing the deleted version directories, if shift has been requested, the directories of versions in grplist from grpend (inclusive) to the end are renamed. The new names are not synthesized but taken from grplist. With idx from grpend to len(grplist) the shifted name of each grplist[idx] is grplist[idx+shift]. In most cases this produces the same result as appending idx+shift to verbase, which is the version group base name derived from the VERSION argument [MostRecentVersion]. However, it is cheaper and somewhat more reliable because the group versions' suffixes may not be monotonic because of explicit user naming instead of the default incremental, in which case grplist indices may not exactly correspond to version suffixes.

To top of [PROGRAM DESIGN]

[Topics]