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.