

# Using P4G.py From The Command Line

J.T. Goldstone

March 4, 2005

## Abstract

This paper presents a shell scripting technique using `p4G.py`, a freely available command line filter. This paper focuses on using this technique in the context of the user client space, though this technique can be used in many contexts where shell scripting is used. The `p4` command provides several useful output formats. Beyond the standard text output and `-Ztag` formats there is the `-G` and `-R` binary formats. These binary formats are easy to parse and readily handle special cases like filenames with spaces. Also, these flags cause some commands (e.g. `groups`) to provide additional output. The `p4G.py` tool parses the `-G` binary output and makes it available to the shell script writer.

## 1 Introduction

The command line interface is a common context for Unix based Perforce users and administrators. Unix-style shells provide quick access to many different commands and through pipelines provide an easy means to process and transform the output of commands. `P4G.py` is a command line filter that helps adapt the `p4` command for use from the command line and from shell scripts.

Currently, Perforce and the user community provide programmatic interfaces for C/C++, Perl, and Ruby users. The `p4G.py` tool exposes some of that power to the shell user.

[Bowles2005] gives a broader view of different techniques and the contexts for scripting Perforce operations. This paper talks about a single shell scripting **technique** used within the **context** of the user workspace. This technique can be customized and combined with techniques as described in [Bowles2005].

## 2 What is P4G.py?

The `p4G.py` is a command line filter; it takes data on its standard input, and writes the results to its standard output. The expected input is a stream of marshaled Python dictionary objects, as produced by the `p4 -G` flag. Typically, `p4G.py` is used in a pipeline right after a `p4 -G` command. The `p4G.py` output is a transformation of the input. The transformation is user controlled through simple printf-style formatting options.

The `p4G.py` tool is a small Python script; it is under 500 lines of code. It is designed to be a general purpose filter of any `p4 -G` output. The script was originally based on the sample Python unmarshaling code in the [P4CMDREF] in the *Global Options* section. The code understands the key naming conventions used by `p4`. Except for large integers, `p4G.py` does not try to interpret the

values in the dictionaries. For large integers <sup>1</sup> it assumes they are the count of seconds since the Unix epoch (00:00:00 GMT Jan. 1, 1970), and formats them with a time stamp.

### 3 Similar Tools

There are other tools that provide similar functionality to `p4G.py`. There are the Perforce libraries for Perl and Ruby. Also, there is the P4Report tool which provides a SQL-like reporting interface to Perforce. P4Report is only available on Windows.

### 4 Getting Started

The basic way to use `p4G.py` is to run `p4 -G` and pipe it to `p4G.py`, and then optionally perform additional filtering. Here are three examples:

```
% p4 -G clients | p4G.py
...
--16--
code: stat
Description: Build Central Build client created by johng
Update: 1107999118 (Wed Feb  9 17:31:58 2005)
Access: 1107999118 (Wed Feb  9 17:31:58 2005)
MapState: 1
Host: build-central
client: johng-builds
Owner: johng
Root: /home/johng/builds
Options: noallwrite noclobber nocompress unlocked nomodtime normdir
...
% p4 -G clients | p4G.py client Owner | sort
...
johng-builds johng
johng-laptop johng
...
% p4 -G clients | p4G.py -k
Access
Description
Host
MapState
Options
Owner
Root
Update
client
code
```

---

<sup>1</sup>Integers larger than 100,000 are considered large. In Unix epoch time, 100,000 seconds is 03:46:40a.m. GMT Jan 2 1970.

The first line formats the output as described in the [P4CMDREF] under *Global Options*. Typically you would use this form to see what keys are available and the format of the values.

The second command line prints only the client name and the owner with a space separator, listing one client per line.

The third command line prints the list of all unique keys found. Typically you might run this form first to see what keys you want to use in command lines of the the second form.

## 5 Why Transform P4 Output?

There are several reasons to transform the regular output of a `p4` command. The transformed `p4` output might be just more readable than the output produced by the `p4` command. This could be due to various factors including what fields are shown, how fields line up, and where carriage returns are inserted. The output of a `p4 opened` command might be easier to read if the user names are at the start of the line and have a fixed width column. This is something `p4G.py` does well.

The other major reason to transform `p4` output is that transformed output can then be easier to process with other commands. The `p4G.py` output can be used as input to other command line filters. The net result of the processing would be to generate some useful report (e.g. how many opened files does each user have) or make some operation easier.

Many Unix-style command line filters are line oriented. They are written with the assumption each record fits on one line. Fields within a record are separated by a separator character. Typically the field separator is the space character or another character like a comma or a colon. The main criteria for a good separator is that data fields do not contain that character.

Transforming Perforce output to fit within this one line per record, and a non-data character separator, makes it easy to work with many Unix-style filters. Regular `p4` output for some commands is one line per record. Those commands typically have a space separator, though many times the fields (e.g. filenames, user comments) themselves contain spaces, making parsing difficult.

## 6 The Secrets of Tagged Output

The tagged output of `p4`, regardless of format (e.g. `-Ztag`, `-R`, or `-G`) used, has additional advantages to the regular output. The previous section talked about the ability to better parse and format tagged output. Another advantage is that some commands report more information than regular output.

An example of a command that has additional output is the `opened` command. The `p4 opened` command only lists the depot pathname for opened files, but `p4 -G` will also list out the client pathname. In the example below, we use the additional output of `p4 opened` and `p4 client -o` to find the local path of an opened file.

```
% p4 opened
//source/proj-wow/ui/Readme#6 - edit default change (kxtext) by chris@chris-wow
% p4 -G opened | p4G.py
--1--
code: stat
rev: 6
clientFile: //chris-dev/ui/Readme
```

```

client: chris-wow
user: chris
action: edit
type: kxtext
depotFile: //source/proj-wow/ui/Readme
change: default
% p4 -G client -o chris-dev | p4G.py Host Root
eng_server /home/chris/dev/projects/wow

```

We can take this example one step further, by trying to find all opened files and their localpaths. Here we use the `join` command to combine the output from the `p4 opened` and `p4 clients` commands. The files are joined using the client names. Then, `sed` is used to clean up the combined output. The generated output does not contain proper Window filenames, but it could be further modified to do that.

```

% p4 -G clients | p4G.py client Host Root | sort > client_host_root
% p4 -G opened -a | p4G.py client clientFile | sort > client_clientfile
% join client_host_root client_clientfile | sed -e 's# //[^/]*##'
...
chris-wow eng_server /home/chris/dev/projects/wow/ui/Readme
...

```

## 7 Dictionary Conventions

Perforce only says that `p4 -G` output generates marshaled Python dictionary objects. A Python dictionary is a data structure that maps keys to values. Dictionaries are basically the same as hash tables or associative arrays in other languages like Perl or AWK. Perforce does not otherwise externally document the format of those dictionaries. Perforce documentation recommends running a command, and looking at the keys generated (e.g. as reported by “`p4G.py -k`”). The following information is based on usage of `p4` versions from 2001.1 to 2004.2. This information might not be fully correct and might change in future releases. Different releases of `p4` may return different sets of keys for the same command against the same server (for example, `groups` behaves differently in release 2004.2).

The `p4 -G` output generates zero or more dictionaries. All the keys of a dictionary are strings, and all the values are strings. The keys and values used depend on the particular `p4` command being run. Key names start with either an upper or lower case letter and sometimes use camel case naming (e.g. `camelCase`), though there is no apparent logic for the capitalization used. Typically a value contains one piece of information (e.g. a depot file name, a user name, the time of an event) and does not require additional parsing. However some commands combine multiple data into a single value, which might require additional parsing like the `sync` command does:

```

% p4 -G sync readme.txt | p4G.py
--1--
code: info
data: //source/readme.txt#3 - added as /home/johng/source/readme.txt
level: 0

```

Some `p4` commands create dictionaries that have fields that can be a list of values (e.g. the files part of a change list, the views of a client spec, etc.) These keys are numbered starting from 0. For example, here's the output of `describe` for a change list with multiple files:

```
% p4 -G describe 1971 | p4G.py

--1--
status: submitted
code: stat
depotFile0: //Infrastructure/scm/trigger/nospaces.pl
depotFile1: //Infrastructure/scm/trigger/submit_limit.pl
action1: delete
action0: delete
client: don-global
user: don
time: 1108424077 (Mon Feb 14 15:34:37 2005)
rev1: 2
rev0: 3
type1: kxtext
type0: kxtext
change: 1971
desc: moved to perforce-admin/server/bin
```

There is less consistency between commands as to the granularity of what a dictionary represents. Some commands put all their data into a single dictionary (such as the `form` commands) other commands place data across several dictionaries.

Below is an example of the `groups` and `group` commands. Here different versions of the `p4` client produce different output for the `groups` command against the same server. The 2004.2 version generates one dictionary per user or subgroup in a group, providing a lot more information than the regular output of `groups`. While the 2003.2 version generates one dictionary per group, without providing user or subgroup information. The last example, shows the `p4 group -o` output, showing that it puts all the form information in one dictionary.

```
% p4.2004.2 -G groups | p4G.py
...
--78--
code: stat
group: dev
isSubGroup: 0
maxResults: 0
user: chris
timeout: 43200
maxScanRows: 0

--79--
code: stat
```

```

group: dev
isSubGroup: 0
maxResults: 0
user: pat
timeout: 43200
maxScanRows: 0
...
% p4.2003.2 -G groups | p4G.py
...
--24--
code: info
data: dev
level: 0
...
% p4 -G group -o dev | p4G.py

--1--
code: stat
Group: dev
Timeout: 43200
MaxResults: unlimited
Users2: chris
Users3: pat
Users0: vance
Users1: bowles
MaxScanRows: unlimited

```

## 8 Form Commands

The `p4G.py` command can read the output of commands that use forms (e.g. `group`, `user`, `change`, etc.). However, there is not much support for transforming that output in way that is useful for input to the same command. Most of these commands have a `-o` flag that outputs the form, and a corresponding `-i` flag that takes a form as input. Typically for the shell user the only support to make automated edits to form data is with a stream editing command like `perl` or `sed` (e.g. `p4 client -o | sed ... | p4 client -i`).

## 9 Common Scripting Building Blocks

There are some common command line filter tools available in a Unix-like environment that can be very useful to use with `p4G.py`. Some of these commands are:

- `grep`
- `perl -p/-n -e`
- `awk/sed/cut`

- `sort`

Here are some useful commands when you want to iterate over the results you get from `p4G.py` command. These commands allow you to quickly run multiple commands, which is a powerful technique. Though, if not done carefully, this can cause the Perforce server to be hammered with commands. Be careful when using these commands. [Bowles2005] has several sections about performance and efficiency which are germane.

- `xargs`
- `foreach`
- `sh`
- `p4 -x`

Here are some useful commands for comparing different results:

- `sort`
- `comm`
- `diff`

## 10 Examples

- **Finding Filenames with Spaces:** Files with spaces in their name are troublesome on some platforms, and easy to create on other platforms. It is possible to use a trigger to prevent their creation. In this situation, we discovered this was a problem and wanted to find existing files with spaces in their names. Here are three solutions, the first uses `p4G.py`.

```
% p4 -G files //... | p4G.py depotFile | grep " " | cat -n
% p4 files //... | sed -e 's/#.*//' | grep " " | cat -n
% p4 -ztag files //... | grep depotFile | sed -e 's#^.*//##' | grep " " | cat -n
```

- **Pretty Printing:** Nicely formatting output can make it more readable. Here is a simple example listing opened files in some more readable formats.

```
% p4 -G opened | p4G.py user depotFile
johng //source/trunk/Makefile
johng //source/trunk/readme.txt
johng //source/trunk/configure
% p4 -G opened | p4G.py -f "%-6s %-6s %s" action user depotFile
delete johng //source/proj-wow/Makefile
edit pat //source/proj-wow/configure
add chris //source/proj-wow/build.xml
```

- **Finding Users without Passwords:** Users without passwords can be a security risk. This example is a solution to an example from [Smith2003]. The `-n NULL` flag tells `p4G.py` to use the string “NULL” for empty strings or unset values.

```
p4 -G users | p4G.py User | xargs -i@ p4 -G user -o @ \
| p4G.py -n NULL User Password | grep NULL
```

- **Finding Old Clients:** Write a script to list the clients in descending access date order (for deleting obsolete clients). The example is another solution to an example from [Smith2003]:

```
p4 -G clients | p4G.py Access client | sort -rn
```

- **Formatting Changes:** The `changes` command by default lists a short description, if you want the full description you have to use the `-l` flag. Though with the `-l` output the description is listed on a separate line. This example shows long output and attempts to put it all on one line. Though if the description contains a carriage return, it will be printed by `p4G.py`. For example, our automated builds use well defined submit comments, we can use this output for further processing.

```
% p4 -G changes -u build -m 5 -l | p4G.py change user client time desc
15416 build build01-bld-a-builder 1109706691 (Tue Mar 1 11:51:31 2005)
Daily Build. Created by build01.ofoto.com. Branch: proc-labinteg.
Label: proclabinteg.393.
...
```

## 11 Design and Future Directions

The `p4G.py` script is designed to be a generic filter of `p4 -G` output. It does not parse the data values (except for integers as time). The goal is that `p4G.py` is not tied to a particular version of Perforce. It currently supports all versions of the `p4` command. Perforce has not committed itself to a particular format of the `-G` output, so future changes could break `p4G.py`. For example, they might start using arrays for values that have multiple values (e.g. the users of a group). Also, as a filter, it is easy to re-filter saved `p4 -G` output, such as from a long running command.

The `p4G.py` tool is a filter and not a wrapper of the `p4` command. There are several design issues with creating a wrapper. The `p4G.py` tool has its own set of options that control its behavior. A wrapped version of `p4` would need to be able to specify options for both the underlying `p4` command and the formatting part. The wrapper could become tied to particular releases of `p4`, as different command line options need to be parsed correctly. As a standalone filter, the `p4G.py` can process saved `p4 -G` output.

It would be possible to create a wrapper. One solution would be for the wrapper to co-opt the `-G` flag, and have it take the options for the filtering part. A wrapper might allow for additional formatting control because it will know what command is being run. Currently `p4G.py` does not know what command produced its input.

The `p4G.py` command does limited processing on the input, only to format whole keys. Additional formatting control might be useful to parse certain values into multiple parts, such as the `data` key returned by a `sync` command.

## 12 References

[Bowles2004] Bowles, J. **Scripting with Perforce**, Proceedings of the 2005 Perforce User's Conference.

[P4CMDREF] **Perforce Command Reference**, Perforce. <http://www.perforce.com/perforce/doc.042/manuals/cmdref/cmdref.pdf>

[P4REPREF] **P4Report User's Guide**, Perforce. <http://www.perforce.com/perforce/doc.042/manuals/p4report/p4report.pdf>

[Smith2003] Smith, T. **Scripting with Perforce Using the Perl and Ruby Interfaces: Tutorial**, Proceedings of the 2003 Perforce User's Conference.\* <http://www.perforce.com/perforce/conf2003/internal/p4script-tutor.pdf>