

Using P4G.py  
From the Command Line

By

J.T. Goldstone

Kodak Information Network/Ofoto

April 15, 2005

# Introduction

- This talk is about a shell scripting **technique**.
- It can be used in multiple **contexts**:
  - Directly from the command line.
  - Within shell scripts.
  - Useful for ad hoc reports and data-mining.
- It will depend on your **purpose**.

# What is P4G.py

- Unix command line filter.
- Python script.
- Based on code in Commands Reference:

```
#!/usr/local/bin/python
import marshal, sys
try:
    num=0
    while 1:
        num=num+1
        print '\n--%d--' % num
        dict = marshal.load(sys.stdin)
        for key in dict.keys(): print "%s: %s" % (key,dict[key])
except EOFError: pass
```

# The Secrets of Keyed Output

- The p4 command has 5 output formats:
  - Plain Text
  - **-s** Prints exit code, and precedes lines with status.
  - **-G** Python binary keyed output
  - **-R** Ruby binary keyed output
  - **-Ztag** (key,value) pairs in text output
  - Advantages of Keyed vs. Plain Text Output
    - Many cases provides additional information
    - Easier to parse

# The “-G” Output

- Binary format.
- Parsed via the standard Python marshal library.
- Consists of a stream of records.
- Records are hash tables (a.k.a associative arrays or dictionaries).
- Data is pre-parsed.

# The "-Ztag" Output

- Output looks like `p4 fstat` format (key.
  - Each field on a separate line.
  - Empty line between records.

```
... depotFile //Source/main/prod/file.c
... rev 1
... change 366
... action add
... type binary
... time 1103242076
```
- Difficult to parse on the command line.
  - Doesn't work well with Unix tools like `grep`.
  - Doesn't work with all commands e.g.: `p4 group -o`
  - Parsing can be broken if data contains an empty line

# Why P4G.py?

- Easier to use than `p4 -Ztag`.
- Provides formatting control.
- Runs on Unix-like platforms.
- Additional information of keyed output.
- Provides fast prototyping/debugging.

# Getting Started

- Has a `-h` help flag to get usage information
- How to use the command:

```
p4 -G clients | p4G.py
```

- Typical Usage pattern use `-k` to find keys

```
p4 -G clients | p4G.py -k
```

```
p4 -G clients | p4G.py client Host Root
```

# The -k/-K Flags

- Use the `-k` flag to get a list of all the keys.
  - Shows numbered keys as `keyN`
  - Shows multi-numbered keys `keyN , N`
- Use the `-K` flag for more details:
  - Shows key summary like `-k`
  - Show count of number of each key
  - Shows minimum/maximum value of indices.
  - Shows common values.

# Sample -K Output

```
% p4 -G groups chris | p4G.py.new -K
3 code v:({'stat': 3})
3 group v:({'admin': 1, 'all': 1,
  'engineers': 1})
3 isSubGroup v:({'0': 3})
3 maxResults v:({'0': 1, '600000': 2})
3 maxScanRows v:({'0': 1, '600000': 2})
3 timeout v:({'43200': 3})
3 user v:({'chris': 3})
3 Records
```

# Numbered Keys

- Used in records with multiple values:
  - A group can have multiple users
  - A client/branch spec can have multiple views
  - A user can list multiple reviews
- P4G.py joins multiple values with a space.

```
p4 -G group -o proj-wow | p4G.py Group Users  
proj-wow alex chris sam
```
- The “-m” flag will print one line per numbered record.

```
p4 -G group -o proj-wow | p4G.py -m Group Users  
proj-wow alex  
proj-wow chris  
proj-wow sam
```

# Printf String Formatting

- The `-f` flag takes a printf-style string.
- Only use the `"%s"` for strings.
- Can use width formats `"%5s"` or `"%-5s"`
- Provides ability to:
  - Add a useful delimiter.
  - Add extra text.

# Time Formatting

- Assumes “large numbers” are Unix epoch time.
- Displays both:
  - Unix epoch time
  - Human readable format.
- Example:
  - 1102630050 (Thu Dec 9 14:07:30 2004)

# Common Scripting Commands

- `grep -i / -v`
- `sed`
- `awk`
- `cut`
- `python | ruby | perl -p / -n -e`

# Scripting Commands for Multiple Items

- `xargs -i`
- `for/foreach` (shell loops)
- `sh/csh` (as a filter)
- `sort`
- `join`
- `comm`

# Offline Usage

- Ability to save command output:

```
p4 -G fstat //Trunk/... > fstat.p4g
```

- Run later with:

```
p4G.py depotFile action < fstat.p4g
```

- Useful when:

- Command takes a long time to run.
- Server is down
- Running disconnected.
- Reduces “trips to the well.”

# Issues with Keyed Output

- Not documented by Perforce:
  - Keys (names and case)
  - Format of values
  - Granularity of a record
- Examples:
  - Multiple data in a single key: p4 sync: **data** key
  - Presence of key means it is true: p4 fstat: **ourLock** key
  - One key tells type of another:
    - p4 groups: **isSubGroup** says if **user** key is subgroup or user
  - Numbered and unnumbered tags with same base name
    - p4 diff2: **depotFile/depotFile2**

# Output Needing Parsing

- Data fields that need parsing

```
% p4 -G sync -f Makefile | p4G.py data
```

```
//Trunk/Readme#146 - refreshing /home/jt/Makefile
```

- A proposed format would be these keys:

```
depotFile: //Trunk/Readme
```

```
rev: 146
```

```
action: refreshing
```

```
clientFile: /home/jt/Makefile
```

- p4 client -o

- View fields need to be parsed.

# Design Considerations

- Filter provides clear separation between what Perforce and p4G.py do.
- No specific knowledge about what keys are or values.
  - Loosely tied to perforce commands & versions.
  - Not totally true: Considers large integers to be Unix epoch time.
- Able to accept input of joined “.p4g” files.
- Could write other tools to generate “.p4g” files.

# Example: Filenames with spaces

- Files with spaces in the name

```
% p4 -G files //... | p4G.py depotFile | grep " "
```

```
//Trunk/projects/wow/Req Doc.doc
```

```
% p4 files //... | sed -e 's!#.*!!' | grep " "
```

```
//Trunk/projects/wow/Req Doc.doc
```

```
% p4 -Ztag files //... | grep depotFile | \
```

```
    sed -e 's!^.*//!//!' | grep " "
```

```
//Trunk/projects/wow/Req Doc.doc
```

# Example:List Opened Files

- Pretty print the list of opened files:

```
% p4 opened
```

```
//Trunk/Makefile#1 - edit default change (xtext) by  
pat@reg-central
```

```
% p4 -G opened | p4G.py user depotFile
```

```
pat //Trunk/Makefile
```

```
chris //Trunk/set.c
```

```
%p4 -G opened|p4G.py -f"%5s %10s %s" action user depotFile
```

```
edit          pat //Trunk/Makefile
```

```
delete       chris //Trunk/set.c
```

# Example: Users Sans Passwords

- Users without passwords:

- Example from Tony Smith's 2003 PUC Perl/Ruby talk.

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

# Example: Sort Clients by Access

- Sort clients in descending order of access date.

– Example from Tony Smith 2003 Talk

```
% p4 -G clients | p4G.py Access client | sort -rn
1112982841 (Fri Apr 8 10:54:01 2005) build1-spec
1112982780 (Fri Apr 8 10:53:00 2005) build2-spec
```

# Example: Find Re-Branched Files

- Example of an ad hoc query:
  - An integration was done incorrectly. Some files were re-branched, that should not have been. Find all re-branched files.

```
% p4 -G describe 1000 > d1000.p4g
```

```
% p4G.py -m action rev depotFile < d1000.p4g | \
  grep branch | cut -c 8- | grep -v "^1"
```

```
33 //Trunk/Makefile
```

```
3 //Trunk/Makeflie
```

# Feature Requests

- Case-insensitive key access.
- More formatting control.
  - Per key formatting, Timestamp Formatting.
- More control for numbered keys.
  - Range syntax (e.g. first 3, last 3, etc)
- Parsing of multi-part fields (like sync).
- Xargs flag.

Using P4G.py  
From the Command Line

By

J.T. Goldstone

Kodak Information Network/Ofoto

April 15, 2005