```
**********************************************************
   12605 Thu Jan 31 09:40:54 2019
new/usr/src/tools/scripts/git-pbchk.py
10328 git pbchk falls over with no changesets
**********************************************************
```

```
#!@TOOLS_PYTHON@
#
#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License version 2
#  as published by the Free Software Foundation.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to the Free Software
#  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
#

#
# Copyright (c) 2008, 2010, Oracle and/or its affiliates. All rights reserved.
# Copyright 2008, 2012 Richard Lowe
# Copyright 2014 Garrett D'Amore <garrett@damore.org>
# Copyright (c) 2015, 2016 by Delphix. All rights reserved.
# Copyright 2016 Nexenta Systems, Inc.
# Copyright 2018 Joyent, Inc.
# Copyright 2018 OmniOS Community Edition (OmniOSce) Association.
#

from __future__ import print_function

import getopt
import io
import os
import re
import subprocess
import sys
import tempfile

if sys.version_info[0] < 3:
    from cStringIO import StringIO
else:
    from io import StringIO

#
# Adjust the load path based on our location and the version of python into
# which it is being loaded.  This assumes the normal onbld directory
# structure, where we are in bin/ and the modules are in
# lib/python(version)?/onbld/Scm/.  If that changes so too must this.
#
sys.path.insert(1, os.path.join(os.path.dirname(__file__), "..", "lib",
                                "python%d.%d" % sys.version_info[:2]))

#
# Add the relative path to usr/src/tools to the load path, such that when run
# from the source tree we use the modules also within the source tree.
#
sys.path.insert(2, os.path.join(os.path.dirname(__file__), ".."))

from onbld.Scm import Ignore
from onbld.Checks import Comments, Copyright, CStyle, HdrChk, WsCheck
from onbld.Checks import JStyle, Keywords, ManLint, Mapfile, SpellCheck

class GitError(Exception):
```



```
    pass

def git(command):
    """Run a command and return a stream containing its stdout (and write its
    stderr to its stdout)"""

    if type(command) != list:
        command = command.split()

    command = ["git"] + command

    try:
        tmpfile = tempfile.TemporaryFile(prefix="git-nits", mode="w+b")
    except EnvironmentError as e:
        raise GitError("Could not create temporary file: %s\n" % e)

    try:
        p = subprocess.Popen(command,
                             stdout=tmpfile,
                             stderr=subprocess.PIPE)
    except OSError as e:
        raise GitError("could not execute %s: %s\n" % (command, e))

    err = p.wait()
    if err != 0:
        raise GitError(p.stderr.read())

    tmpfile.seek(0)
    lines = []
    for l in tmpfile:
        lines.append(l.decode('utf-8', 'replace'))
    return lines

def git_root():
    """Return the root of the current git workspace"""

    p = git('rev-parse --git-dir')
    dir = p[0]

    return os.path.abspath(os.path.join(dir, os.path.pardir))

def git_branch():
    """Return the current git branch"""

    p = git('branch')

    for elt in p:
        if elt[0] == '*':
            if elt.endswith('(no branch)'):
                return None
            return elt.split()[1]

def git_parent_branch(branch):
    """Return the parent of the current git branch.

    If this branch tracks a remote branch, return the remote branch which is
    tracked.  If not, default to origin/master."""

    if not branch:
        return None

    p = git(["for-each-ref", "--format=%(refname:short) %(upstream:short)",
             "refs/heads/"])

    if not p:
        sys.stderr.write("Failed finding git parent branch\n")
```

```
128         sys.exit(1)
128         sys.exit(err)

130     for line in p:
131         # Git 1.7 will leave a ' ' trailing any non-tracking branch
132         if ' ' in line and not line.endswith(' \n'):
133             local, remote = line.split()
134             if local == branch:
135                 return remote
136     return 'origin/master'

138 def git_comments(parent):
139     """Return a list of any checkin comments on this git branch"""

141     p = git('log --pretty=tformat:%%B:SEP: %s..' % parent)

143     if not p:
144         sys.stderr.write("No outgoing changesets found - missing -p option?\n");
145         sys.exit(1)
144         sys.stderr.write("Failed getting git comments\n")
145         sys.exit(err)

147     return [x.strip() for x in p if x != ':SEP:\n']

149 def git_file_list(parent, paths=None):
150     """Return the set of files which have ever changed on this branch.

152     NB: This includes files which no longer exist, or no longer actually
153     differ."""

155     p = git("log --name-only --pretty=format: %s.. %s" %
156             (parent, ' '.join(paths)))

158     if not p:
159         sys.stderr.write("Failed building file-list from git\n")
160         sys.exit(1)
160         sys.exit(err)

162     ret = set()
163     for fname in p:
164         if fname and not fname.isspace() and fname not in ret:
165             ret.add(fname.strip())

167     return ret

169 def not_check(root, cmd):
170     """Return a function which returns True if a file given as an argument
171     should be excluded from the check named by 'cmd'"""

173     ignorefiles = list(filter(os.path.exists,
174                               [os.path.join(root, ".git", "%s.NOT" % cmd),
175                                os.path.join(root, "exception_lists", cmd)]))
176     return Ignore.ignore(root, ignorefiles)

178 def gen_files(root, parent, paths, exclude):
179     """Return a function producing file names, relative to the current
180     directory, of any file changed on this branch (limited to 'paths' if
181     requested), and excluding files for which exclude returns a true value """

183     # Taken entirely from Python 2.6's os.path.relpath which we would use if we
184     # could.
185     def relpath(path, here):
186         c = os.path.abspath(os.path.join(root, path)).split(os.path.sep)
187         s = os.path.abspath(here).split(os.path.sep)
188         l = len(os.path.commonprefix((s, c)))
189         return os.path.join(*[os.path.pardir] * (len(s)-l) + c[l:])
```



```
191     def ret(select=None):
192         if not select:
193             select = lambda x: True

195         for abspath in git_file_list(parent, paths):
196             path = relpath(abspath, '.')
197             try:
198                 res = git("diff %s HEAD %s" % (parent, path))
199             except GitError as e:
200                 # This ignores all the errors that can be thrown. Usually, this
201                 # means that git returned non-zero because the file doesn't
202                 # exist, but it could also fail if git can't create a new file
203                 # or it can't be executed.  Such errors are 1) unlikely, and 2)
204                 # will be caught by other invocations of git().
205                 continue
206             empty = not res
207             if (os.path.isfile(path) and not empty and
208                 select(path) and not exclude(abspath)):
209                 yield path
210     return ret

212 def comchk(root, parent, flist, output):
213     output.write("Comments:\n")

215     return Comments.comchk(git_comments(parent), check_db=True,
216                            output=output)


219 def mapfilechk(root, parent, flist, output):
220     ret = 0

222     # We are interested in examining any file that has the following
223     # in its final path segment:
224     #     - Contains the word 'mapfile'
225     #     - Begins with 'map.'
226     #     - Ends with '.map'
227     # We don't want to match unless these things occur in final path segment
228     # because directory names with these strings don't indicate a mapfile.
229     # We also ignore files with suffixes that tell us that the files
230     # are not mapfiles.
231     MapfileRE = re.compile(r'.*((mapfile[^/]*)|(/map\.+[^/]*)|(\.map))$',
232         re.IGNORECASE)
233     NotMapSuffixRE = re.compile(r'.*\.[ch]$', re.IGNORECASE)

235     output.write("Mapfile comments:\n")

237     for f in flist(lambda x: MapfileRE.match(x) and not
238                    NotMapSuffixRE.match(x)):
239         with io.open(f, encoding='utf-8', errors='replace') as fh:
240             ret |= Mapfile.mapfilechk(fh, output=output)
241     return ret

243 def copyright(root, parent, flist, output):
244     ret = 0
245     output.write("Copyrights:\n")
246     for f in flist():
247         with io.open(f, encoding='utf-8', errors='replace') as fh:
248             ret |= Copyright.copyright(fh, output=output)
249     return ret

251 def hdrchk(root, parent, flist, output):
252     ret = 0
253     output.write("Header format:\n")
254     for f in flist(lambda x: x.endswith('.h')):
255         with io.open(f, encoding='utf-8', errors='replace') as fh:
```

```
256             ret |= HdrChk.hdrchk(fh, lenient=True, output=output)
257     return ret

259 def cstyle(root, parent, flist, output):
260     ret = 0
261     output.write("C style:\n")
262     for f in flist(lambda x: x.endswith('.c') or x.endswith('.h')):
263         with io.open(f, encoding='utf-8', errors='replace') as fh:
264             ret |= CStyle.cstyle(fh, output=output, picky=True,
265                                  check_posix_types=True,
266                                  check_continuation=True)
267     return ret

269 def jstyle(root, parent, flist, output):
270     ret = 0
271     output.write("Java style:\n")
272     for f in flist(lambda x: x.endswith('.java')):
273         with io.open(f, encoding='utf-8', errors='replace') as fh:
274             ret |= JStyle.jstyle(fh, output=output, picky=True)
275     return ret

277 def manlint(root, parent, flist, output):
278     ret = 0
279     output.write("Man page format/spelling:\n")
280     ManfileRE = re.compile(r'.*\.[0-9][a-z]*$', re.IGNORECASE)
281     for f in flist(lambda x: ManfileRE.match(x)):
282         with io.open(f, encoding='utf-8', errors='replace') as fh:
283             ret |= ManLint.manlint(fh, output=output, picky=True)
284             ret |= SpellCheck.spellcheck(fh, output=output)
285     return ret

287 def keywords(root, parent, flist, output):
288     ret = 0
289     output.write("SCCS Keywords:\n")
290     for f in flist():
291         with io.open(f, encoding='utf-8', errors='replace') as fh:
292             ret |= Keywords.keywords(fh, output=output)
293     return ret

295 def wscheck(root, parent, flist, output):
296     ret = 0
297     output.write("white space nits:\n")
298     for f in flist():
299         with io.open(f, encoding='utf-8', errors='replace') as fh:
300             ret |= WsCheck.wscheck(fh, output=output)
301     return ret

303 def run_checks(root, parent, cmds, paths='', opts={}):
304     """Run the checks given in 'cmds', expected to have well-known signatures,
305     and report results for any which fail.

307     Return failure if any of them did.

309     NB: the function name of the commands passed in is used to name the NOT
310     file which excepts files from them."""

312     ret = 0

314     for cmd in cmds:
315         s = StringIO()

317         exclude = not_check(root, cmd.__name__)
318         result = cmd(root, parent, gen_files(root, parent, paths, exclude),
319                      output=s)
320         ret |= result
```



```
322         if result != 0:
323             print(s.getvalue())

325     return ret

327 def nits(root, parent, paths):
328     cmds = [copyright,
329             cstyle,
330             hdrchk,
331             jstyle,
332             keywords,
333             manlint,
334             mapfilechk,
335             wscheck]
336     run_checks(root, parent, cmds, paths)

338 def pbchk(root, parent, paths):
339     cmds = [comchk,
340             copyright,
341             cstyle,
342             hdrchk,
343             jstyle,
344             keywords,
345             manlint,
346             mapfilechk,
347             wscheck]
348     run_checks(root, parent, cmds)

350 def main(cmd, args):
351     parent_branch = None
352     checkname = None

354     try:
355         opts, args = getopt.getopt(args, 'b:c:p:')
356     except getopt.GetoptError as e:
357         sys.stderr.write(str(e) + '\n')
358         sys.stderr.write("Usage: %s [-c check] [-p branch] [path...]\n" % cmd)
359         sys.exit(1)

361     for opt, arg in opts:
362         # We accept "-b" as an alias of "-p" for backwards compatibility.
363         if opt == '-p' or opt == '-b':
364             parent_branch = arg
365         elif opt == '-c':
366             checkname = arg

368     if not parent_branch:
369         parent_branch = git_parent_branch(git_branch())

371     if checkname is None:
372         if cmd == 'git-pbchk':
373             checkname = 'pbchk'
374         else:
375             checkname = 'nits'

377     if checkname == 'pbchk':
378         if args:
379             sys.stderr.write("only complete workspaces may be pbchk'd\n");
380             sys.exit(1)
381         pbchk(git_root(), parent_branch, None)
382     elif checkname == 'nits':
383         nits(git_root(), parent_branch, args)
384     else:
385         run_checks(git_root(), parent_branch, [eval(checkname)], args)

387 if __name__ == '__main__':
```

```
388     try:
389         main(os.path.basename(sys.argv[0]), sys.argv[1:])
390     except GitError as e:
391         sys.stderr.write("failed to run git:\n %s\n" % str(e))
392         sys.exit(1)
```