```
**********************************************************
   12328 Thu Oct  4 15:52:12 2018
new/usr/src/tools/scripts/git-pbchk.py
9867 pbchk exception_lists only work from top srcdir
**********************************************************
```

```
#!@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.
#

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

from cStringIO 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")
    except EnvironmentError, e:
        raise GitError("Could not create temporary file: %s\n" % e)

    try:
        p = subprocess.Popen(command,
                             stdout=tmpfile,
                             stderr=subprocess.PIPE)
    except OSError, 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)
    return tmpfile


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

    p = git('rev-parse --git-dir')

    if not p:
        sys.stderr.write("Failed finding git workspace\n")
        sys.exit(err)

    return os.path.abspath(os.path.join(p.readlines()[0],
                                        os.path.pardir))


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

    p = git('branch')

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

    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/"])
```

```
128     if not p:
129         sys.stderr.write("Failed finding git parent branch\n")
130         sys.exit(err)

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


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

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

146     if not p:
147         sys.stderr.write("Failed getting git comments\n")
148         sys.exit(err)

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


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

156     NB: This includes files which no longer exist, or no longer actually
157     differ."""

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

162     if not p:
163         sys.stderr.write("Failed building file-list from git\n")
164         sys.exit(err)

166     ret = set()
167     for fname in p:
168         if fname and not fname.isspace() and fname not in ret:
169             ret.add(fname.strip())

171     return ret


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

178     ignorefiles = filter(os.path.exists,
179                          [os.path.join(root, ".git", "%s.NOT" % cmd),
180                           os.path.join(root, "exception_lists", cmd)])
181     return Ignore.ignore(root, ignorefiles)


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

189     # Taken entirely from Python 2.6's os.path.relpath which we would use if we
190     # could.
191     def relpath(path, here):
192         c = os.path.abspath(os.path.join(root, path)).split(os.path.sep)
193         s = os.path.abspath(here).split(os.path.sep)
```



```
194         l = len(os.path.commonprefix((s, c)))
195         return os.path.join(*[os.path.pardir] * (len(s)-l) + c[l:])

197     def ret(select=None):
198         if not select:
199             select = lambda x: True

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


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

222     return Comments.comchk(git_comments(parent), check_db=True,
223                            output=output)


226 def mapfilechk(root, parent, flist, output):
227     ret = 0

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

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

244     for f in flist(lambda x: MapfileRE.match(x) and not
245                    NotMapSuffixRE.match(x)):
246         fh = open(f, 'r')
247         ret |= Mapfile.mapfilechk(fh, output=output)
248         fh.close()
249     return ret
```

```
252 def copyright(root, parent, flist, output):
253     ret = 0
254     output.write("Copyrights:\n")
255     for f in flist():
256         fh = open(f, 'r')
257         ret |= Copyright.copyright(fh, output=output)
258         fh.close()
259     return ret


262 def hdrchk(root, parent, flist, output):
263     ret = 0
264     output.write("Header format:\n")
265     for f in flist(lambda x: x.endswith('.h')):
266         fh = open(f, 'r')
267         ret |= HdrChk.hdrchk(fh, lenient=True, output=output)
268         fh.close()
269     return ret


272 def cstyle(root, parent, flist, output):
273     ret = 0
274     output.write("C style:\n")
275     for f in flist(lambda x: x.endswith('.c') or x.endswith('.h')):
276         fh = open(f, 'r')
277         ret |= CStyle.cstyle(fh, output=output, picky=True,
278                              check_posix_types=True,
279                              check_continuation=True)
280         fh.close()
281     return ret


284 def jstyle(root, parent, flist, output):
285     ret = 0
286     output.write("Java style:\n")
287     for f in flist(lambda x: x.endswith('.java')):
288         fh = open(f, 'r')
289         ret |= JStyle.jstyle(fh, output=output, picky=True)
290         fh.close()
291     return ret


294 def manlint(root, parent, flist, output):
295     ret = 0
296     output.write("Man page format/spelling:\n")
297     ManfileRE = re.compile(r'.*\.[0-9][a-z]*$', re.IGNORECASE)
298     for f in flist(lambda x: ManfileRE.match(x)):
299         fh = open(f, 'r')
300         ret |= ManLint.manlint(fh, output=output, picky=True)
301         ret |= SpellCheck.spellcheck(fh, output=output)
302         fh.close()
303     return ret

305 def keywords(root, parent, flist, output):
306     ret = 0
307     output.write("SCCS Keywords:\n")
308     for f in flist():
309         fh = open(f, 'r')
310         ret |= Keywords.keywords(fh, output=output)
311         fh.close()
312     return ret

314 def wscheck(root, parent, flist, output):
315     ret = 0
```



```
316     output.write("white space nits:\n")
317     for f in flist():
318         fh = open(f, 'r')
319         ret |= WsCheck.wscheck(fh, output=output)
320         fh.close()
321     return ret

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

327     Return failure if any of them did.

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

332     ret = 0

334     for cmd in cmds:
335         s = StringIO()

337         exclude = not_check(root, cmd.func_name)
338         result = cmd(root, parent, gen_files(root, parent, paths, exclude),
339                      output=s)
340         ret |= result

342         if result != 0:
343             print s.getvalue()

345     return ret


348 def nits(root, parent, paths):
349     cmds = [copyright,
350             cstyle,
351             hdrchk,
352             jstyle,
353             keywords,
354             manlint,
355             mapfilechk,
356             wscheck]
357     run_checks(root, parent, cmds, paths)


360 def pbchk(root, parent, paths):
361     cmds = [comchk,
362             copyright,
363             cstyle,
364             hdrchk,
365             jstyle,
366             keywords,
367             manlint,
368             mapfilechk,
369             wscheck]
370     run_checks(root, parent, cmds)


373 def main(cmd, args):
374     parent_branch = None
375     checkname = None

377     try:
378         opts, args = getopt.getopt(args, 'c:p:')
379     except getopt.GetoptError, e:
380         sys.stderr.write(str(e) + '\n')
381         sys.stderr.write("Usage: %s [-c check] [-p branch] [path...]\n" % cmd)
```

```
382         sys.exit(1)

384     for opt, arg in opts:
385         # backwards compatibility
386         if opt == '-b':
387             parent_branch = arg
388         elif opt == '-c':
389             checkname = arg
390         elif opt == '-p':
391             parent_branch = arg

393     if not parent_branch:
394         parent_branch = git_parent_branch(git_branch())

396     if checkname is None:
397         if cmd == 'git-pbchk':
398             checkname= 'pbchk'
399         else:
400             checkname = 'nits'

402     if checkname == 'pbchk':
403         if args:
404             sys.stderr.write("only complete workspaces may be pbchk'd\n");
405             sys.exit(1)
406         pbchk(git_root(), parent_branch, None)
407     elif checkname == 'nits':
408         nits(git_root(), parent_branch, args)
409     else:
410         run_checks(git_root(), parent_branch, [eval(checkname)], args)

412 if __name__ == '__main__':
413     try:
414         main(os.path.basename(sys.argv[0]), sys.argv[1:])
415     except GitError, e:
416         sys.stderr.write("failed to run git:\n %s\n" % str(e))
417         sys.exit(1)
```