Python Errors When Migrating from v2 to v3

Charles E. Matthews
Fifth Generation Systems, Ltd.
April 8, 2021

Software failure diagnosis is a complex task. However, there are times when this task complexity can be significantly reduced. One example when diagnostic tasks are simpler than normal is when the application design remains unchanged, but errors occur because the underlying language compiler/interpreter undergoes a version change. This report documents errors in a software package that resulted from changing the version of Python that executed the application. There was no change to the application design. Only the Python version change caused the application errors. It is instructive to analyze the errors that occurred and the methods that one can use to find them efficiently.

By analyzing the types of errors that occurred, a clever developer could devise ways to improve the diagnostic task for other environmental changes as well. Changes that could cause similar types of errors are: a) the version of a component library changes, b) the interface for a connected device changes, c) the application migrates to a different platform, etc.

Application Description

This application is an open source application – the **Repo** package. This application consists of 36 Python files in two directories. It is a component of the **AOSP** – the Android Open Source Project. Before a developer can build an Android image for a target hardware platform, he must download numerous files from various git repositories. This build process is an extremely complex task. The **Repo** component manages some of the complexity in interacting with the various git repositories.

During the build process, an initial Bash script downloads a number of Python files and an executive Bash script[1] for **Repo**. The executive controller calls the **Repo** package multiple times to perform various executive functions.

Error Categories

The errors fall into three categories that indicate the degree of difficulty that it took to diagnose and fix the error. The Python interpreter immediately identified some syntax errors as failures as soon as it read the Python file. Because the Python executive **interprets** rather than **compiles** a source file, all processing ends when the first error occurs. The debug task then becomes a sequence of "run the app", "fix the resulting error", and "repeat to find the next error". Consequently, the debug task became more

[1] The executive Bash script performs a check on the versions of Python2 and Python3 that are installed on the development machine. The script then determines whether the **Repo** component is executed with Python2 or Python3. If the script chooses Python2 to execute the **Repo** files, it sends multiple warning messages to the user – "*repo: warning: Python 2 is no longer supported; Please upgrade to Python 3.6+*". These messages are examples of an unfortunate paradox because Python3 forces the **Repo** execution to fail.

efficient by identifying the coding pattern that caused the syntax error and searching all the files for similar statement patterns. The syntax errors in this category are simple command patterns that are easily searchable.

Another category of errors were also simple syntax errors, but the interpreter did not identify them as a failure until it executed the code. These errors have a slightly higher complexity than the first category because they will not cause a failure if the specific command path is not executed. Because command paths through a software application are logic dependent, some errors will not be flagged by the interpreter if its statement path is not taken. Fortunately, most of these errors also have a simple syntax that is easily searchable. One of the errors in this category, B4 (see the following section), is easily searchable, but it is not instantly apparent whether the command pattern is a real error or is a correct sequence. The developer must do additional analysis to determine whether the code is truly an error or not.

The third category of errors are command sequences that appear to be syntactically correct but result in incorrect operation. Although these errors require a more detailed diagnosis to identify their solution, the number of errors in this category is relatively small – only 7.4% of the total errors.

Errors that are immediate interpreter failures.

The Python interpreter immediately flags these commands as errors as soon as it reads the source file. The fundamental cause for these errors is command syntax changes between Python2 and Python3. The correction for syntactical errors is usually apparent, and the developer can quickly search all application files for all locations of the error. Searching the files for other error locations is advisable because the interpreter stops execution as soon as an error occurs. Therefore, only the first error occurrence is flagged by the interpreter and all other occurrences are hidden until the developer initiates the next run after a software update. 22.2% of the errors are in this category.

A1: Exception catches

```
try:
    cmd.Execute(copts, cargs)
~~except ManifestInvalidRevisionError, e:~~
except ManifestInvalidRevisionError as e:
    print >>sys.stderr, 'error: %s' % str(e)
    sys.exit(1)
```

Count = 22

A2: Change syntax to raise exceptions

```
def _CurrentWrapperVersion():
    VERSION = None
    pat = re.compile(r'^VERSION *=')
    fd = open(_MyWrapperPath())
    for line in fd:
        if pat.match(line):
            fd.close()
            exec line
            return VERSION
    ~~raise NameError, 'No VERSION in repo script'~~
    raise NameError('No VERSION in repo script')
```

Count = 7

A3: Change **xrange** operator to **range**

```
~~for i in xrange(0, len(argv)):~~
for i in range(0, len(argv)):
    if not argv[i].startswith('-'):
        name = argv[i]
        if i > 0:
            glob = argv[:i]
        argv = argv[i + 1:]
        break
```

Count = 3

A4: Change dictionary iterator from **.iteritems()** to **.items()**

```
~~for name, id in all.iteritems():~~
for name, id in all.items():
    if name.startswith(R_HEADS):
        name = name[len(R_HEADS):]
        b = self.GetBranch(name)
        b.current = name == current
        b.published = None
        b.revision = id
        heads[name] = b
```

Count = 10

A5: Change **import urllib2** to **import urllib.request** or **import urllib.error**

```
~~import urllib2~~
import urllib.request
import urllib.error
```

Count = 2

A6: Change **import cPickle**

```
~~import cPickle~~
import pickle as cPickle
```

Count = 1

A7: Change **import StringIO** to **from io import StringIO**

```
~~import StringIO~~
from io import StringIO
```

Count = 1

A8: Change **import xmlrpclib** to **import xmlrpc.client**

```
~~import xmlrpclib~~
import xmlrpc.client
```

Count = 1

A9: When importing from a package, specify the package folder

```
~~from sync import Sync~~
from subcmds.sync import Sync
```

Count = 1

Errors that cause an interpreter failure when the statement is executed.

The interpreter flags these errors not when it reads the source file but when it executes the command. Therefore the identification of these errors by the interpreter is path dependent. If the error occurs in a statement path that is not executed, the interpreter does not identify the error. Therefore the diagnostic task is slightly more complex than the first category. However, because these errors are also syntax errors, the developer can easily correct them, and he can easily search the source files for other occurrences of the error. Again, the primary cause for these errors is syntax changes between Python2 and Python3. 70.4% of the errors are in this category.

B1: Change syntax for **print** statements

```
~~print ''~~
~~print 'repo %s initialized in %s' % (type, self.manifest.topdir)~~
print('')
print('repo %s initialized in %s' % (type, self.manifest.topdir))
```

Count = 35

B2: Change syntax for **print** statements that redirect to an output stream

```
def Sync_NetworkHalf(self, quiet=False):
  """Perform only the network IO portion of the sync process.
     Local working directory/branch state is not affected.
  """
  is_new = not self.Exists
  if is_new:
    if not quiet:
      ~~print >>sys.stderr~~
      ~~print >>sys.stderr, 'Initializing project %s ...' % self.name~~
      print('', file=sys.stderr)
      print('Initializing project %s ...' % self.name,
            file=sys.stderr)
    self._InitGitDir()
```
Count = 104

B3: **GitConfig._cache_dict** changed from strings to byte arrays

```
~~v = self._cache[_key(name)]~~
v = self._cache[_key(name).encode()]
```
Count = 10

B4: Operators on byte arrays vs strings

The solution for this error is one of two options, but the choice is not readily apparent without some additional analysis. Either the target object is supposed to be a string, or it is supposed to be a byte array. If the target variable is supposed to be a string but is a byte array, then find the command that sets the target variable and correct it. If the target variable is supposed to be a byte array, then change the command to work with byte arrays.

```
for name in self._cache.keys():
    ~~p = name.split('.')~~
    p = name.split(b'.')
    if 2 == len(p):
        section = p[0]
        subsect = ''
    else:
        section = p[0]
        ~~subsect = '.'.join(p[1:-1])~~
        subsect = b'.'.join(p[1:-1])
    if section not in d:
        d[section] = set()
    d[section].add(subsect)
    self._section_dict = d
```
Count = 3

Errors that were caused by language changes from Python2 to Python3.

The interpreter does not flag these errors either when it reads the source file or when it executes the source code. The source code causes incorrect behavior, and the developer must first recognize the incorrect behavior and then correct the source code to produce correct behavior. None of the corrections for these errors required a design change for the application. The developer can search the repository of code for similar code patterns, but it is not easily apparent whether the pattern requires a correction. The developer must do additional analysis before changing the code. 7.4% of the errors are in this category.

C1: Replace **exec** statement

This block of code searches a file for a statement like "*VERSION = (2, 8)*". If it occurs, the **exec** command executes the line and updates the **VERSION** variable in the Python code. This command works in Python2 but not in Python3. There may be a scoping issue in Python3 that prevents the Python variable from being assigned. The solution required an alternate set of commands.

```
Def _CurrentWrapperVersion():
    VERSION = None
    pat = re.compile(r'^VERSION *=')
    fd = open(_MyWrapperPath())
    for line in fd:
        if pat.match(line):
            fd.close()
            ~~exec line~~
            line = line[line.index("=") + 1:].strip("() \n\r")
            VERSION = tuple(map(int, line.split(", ")))
            return VERSION
    raise NameError, 'No VERSION in repo script'
```

Count = 1

C2: Add "*universal_newlines = True*" to **subprocess.Popen** call

It appears that Python2 opens the process pipe in text mode by default. With Python3, the additional parameter is required to open the pipe in text mode.

```
p = subprocess.Popen(command,
                     cwd = cwd,
                     env = env,
                     universal_newlines = True,
                     stdin = stdin,
                     stdout = stdout,
                     stderr = stderr)
```

Count = 1

C3: Open files in text mode instead of binary mode

Python2 appears to open files in text mode even though the file mode parameter indicates binary mode. With Python3, the file mode must not be set for binary mode if text mode is desired.

```
def _ReadPackedRefs(self):
    path = os.path.join(self._gitdir, 'packed-refs')
    try:
        ~~fd = open(path, 'rb')~~
        fd = open(path, 'r')
        mtime = os.path.getmtime(path)
    except IOError:
        return
    except OSError:
        return
    try:
        for line in fd:
            if line[0] == '#':
                continue
            if line[0] == '^':
                continue

            line = line[:-1]
            p = line.split(' ')
            id = p[0]
            name = p[1]

            self._phyref[name] = id
    finally:
        fd.close()
    self._mtime['packed-refs'] = mtime
```

Count = 5

C4: When writing directly to **sys.stdout**, flush the buffer before prompting for user input
With Python2, any write to **sys.stdout** completed before a subsequent read from **sys.stdin** occurred. With Python3, it necessary to flush the **sys.stdout** buffer before reading the **sys.stdin** stream.

```
def _Prompt(self, prompt, value):
    mp = self.manifest.manifestProject

    sys.stdout.write('%-10s [%s]: ' % (prompt, value))
    sys.stdout.flush()
    a = sys.stdin.readline().strip()
    if a == '':
      return value
    return a
```

Count = 6

C5: Change in operation of **map** command

It appears that the **map** command changed substantially between Python2 and Python3. In Python3, the **map** command returns a **map** iterator. Amending the code to produce correct results was non-intuitive.

```
class Remote(object):
  """Configuration options related to a remote.
  """
  def __init__(self, config, name):
    self._config = config
    self.name = name
    self.url = self._Get('url')
    self.review = self._Get('review')
    self.projectname = self._Get('projectname')
    ~~self.fetch = map(lambda x: RefSpec.FromString(x),~~
                     ~~self._Get('fetch', all=True))~~
    if len(self._Get('fetch', all=True)) == 0:
      self.fetch = []
    else:
      self.fetch = [RefSpec.FromString(self._Get('fetch', all=True))]
    self._review_protocol = None
```

Count = 3

Error Details

When executed with Python3, the Repo package has 18 errors that occur 216 times. Of the 36 Python files, only 5 of them are error free. The file **sync.py** has the most number of errors (30), but the file **git_config.py** has the most number of unique errors (9). The following tables tabulate the errors found in each file.

	A1	A2	A3	A4	A5	A6	A7	A8	A9
color.py									
command.py									
editor.py	1								
error.py									
git_command.py	1								
git_config.py	3		1		1	1			
git_refs.py				2					
main.py	4	1	1						
manifest_xml.py	1	5							
pager.py	1								
progress.py									
project.py	7			7	1				
trace.py									
__init__.py		1							

abandon.py									
branches.py				1					
checkout.py									
cherry_pick.py									
diff.py									
download.py									
forall.py									
grep.py									
help.py									
init.py	1								
list.py									
manifest.py									
prune.py									
rebase.py									
selfupdate.py									
smartsync.py									1
stage.py			1						
start.py									
status.py							1		
sync.py	1							1	
upload.py	2								
version.py									
	22	7	3	10	2	1	1	1	1

	B1	B2	B3	B4
color.py				
command.py				
editor.py	1	1		
error.py				
git_command.py		2		
git_config.py		1	10	2
git_refs.py				
main.py		12		
manifest_xml.py				
pager.py		1		
progress.py				
project.py	2	6		1
trace.py		1		
__init__.py				

abandon.py		4		
branches.py		1		
checkout.py		2		
cherry_pick.py		8		
diff.py				
download.py		4		
forall.py		1		
grep.py	1	2		
help.py	8	1		
init.py	7	9		
list.py	1			
manifest.py		3		
prune.py	1			
rebase.py		4		
selfupdate.py		1		
smartsync.py				
stage.py	1	1		
start.py		3		
status.py	2			
sync.py	1	27		
upload.py	6	9		
version.py	4			
	35	104	10	3

	C1	C2	C3	C4	C5	
color.py						0
command.py						0
editor.py						3
error.py						0
git_command.py		1				4
git_config.py			1		2	22
git_refs.py			2			4
main.py	1					19
manifest_xml.py						6
pager.py						2
progress.py						0
project.py			2		1	27
trace.py						1
__init__.py						1

abandon.py						4
branches.py						2
checkout.py						2
cherry_pick.py						8
diff.py						0
download.py						4
forall.py						1
grep.py						3
help.py						9
init.py				3		20
list.py						1
manifest.py						3
prune.py						1
rebase.py						4
selfupdate.py						1
smartsync.py						1
stage.py						3
start.py						3
status.py				1		4
sync.py						30
upload.py				2		19
version.py						4
	1	1	5	6	3	216

Lessons Learned

Software fault diagnosis is a complex cognitive task. Under normal circumstances, software faults are unique from each other. However, it quickly became apparent that these faults were not unique. Many faults were strongly similar to each other. When a developer observes patterns of similarity, one can reduce the complexity of fault diagnosis by searching for other occurrences of the pattern. This action reduces the amount of time required to produce a fault-free system.

This report documents the errors that occurred as a result of using the Python v3 compiler instead of Python v2. Neither the application design nor its basic implementation changed. The errors, by themselves, are not extremely important. What is important is the observation that many of the errors appeared to be related. Because they appeared to be related, a couple of useful questions are, "*Why are they related?*" and "*Can the developer take advantage of the similarities?*"

In this case, these errors are similar because their root cause is due to a change that is unrelated to the application design or implementation – the compiler version change. This version change resulted in groups of faults that were similar. By identifying the fault pattern, the developer can search the source code for other occurrences of the pattern to identify other faults that have not yet been identified. This reduces the total time that is necessary to produce a working system.

Although the Python compiler version change is the root cause for these errors, it is not the only change that will produce fault patterns. An observant developer should keep this idea in mind when analyzing faults and exploit any fault patterns when they are found.