

```
#!/usr/bin/env python
```

```
"""pycount.py -- A very initial effort to Python code metrics.
```

```
This program started as a hack, bending and twisting pylint.py by Tim Peters. At that time I was interested on code metrics and thought let's do something quick and dirty. pycount then and today scans a Python file and reports back various categories of lines it thinks it has found, like comments, doc strings, blank lines and, sometimes, real code as well.
```

```
The output can be either as a table with only the number of lines found for each file or as a listing of the file(s) prefixed with some running numbers and classification for each line.
```

```
The former is useful to scan a whole project e.g. when you need to know if the project is documented well or at all and where this info can be found. The latter is at least a nice new view to your own sources or that of others if nothing else!
```

```
There are a couple of minor known bugs with pycount like: Doc strings must be tripple-quoted ones otherwise they are classified as normal source code. Continuation lines ending with a backslash are not treated at all. Complex regular expressions (as in pycount itself) can knock the parser down, quickly. There is a built-in quick-and-dirty solution to this which might work whenever the problem is on one line only. But in "most cases" it works...
```

```
Usage:
```

```
pycount.py [-v] <file1> [<file2> ...]
pycount.py [-v] <expr>
pycount.py [-F] <linetypes> <file>
pycount.py [-R] <expr>
```

```
where <fileN>      is a Python file (usually ending in .py),
      <linetypes>  is a comma-seperated list of python line
                    type codes (code, comment, doc string, blank)
                    e.g. '###' or 'DOC,###,---'
      <expr>       is a shell expression with meta-characters
                    (note that for -R you must quote it)
      -v          verbose flag, listing the source
      -F          filter flag, listing the filtered source
      -R          apply recursively on subdirectories
```

```
NOTES:
```

```
TODO:
```

- Don't filter first line if '#!<path> python'(?).
- De-obfuscate top-level if-stmt in main().
- Improve usage as as a module.
- Print statistics as percentage figures, maybe.
- Write some test cases using pyunit.

```
DONE:
```

- Replace Unix 'find' with Python os.path.walk / fnmatch.
- Scanning should also work recursively (-R option).
- Test stdin case for single files.
- Return total count per category when run on multiple files.
- Add a feature to uncomment files.

```
HISTORY:
```

- 0.0.1 : 1997-??-?? : copy/past from Tim Peter's pylint
- 0.0.2 : 1997-??-?? : included some refinements by Tim

- 0.0.3 : 1997-07-22 : doc & (C) (borrowed from M.-A. Lemburg)
- 0.0.4 : 1998-08-25 : replaced regex/regsub with re module,
added a global line counter in -v mode
- 0.0.5 : 1998-11-25 : code embellishments, recursive on files, ...
- 0.0.6 : 2000-07-04 : fixed typos, improved doc

FUTURE:

- The future is always uncertain...

(c) Copyright by Dinu C. Gherman, 1998, gherman@europemail.com

Permission to use, copy, modify, and distribute this software and its documentation without fee and for any purpose, except direct commercial advantage, is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

THE AUTHOR DINU C. GHERMAN DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE!

Dinu C. Gherman,

2000-07-04

"""

__version__ = 0,0,6

```
import sys
import os
import re
import string
import getopt
import fnmatch
```

```
# Compile helper regular expressions.
```

```
# Reg. exps. to find the end of a triple quote, given that
# we know we're in one; use the "match" method; .span()[1]
# will be the index of the character following the final
# quote.
```

```
_squote3_finder = re.compile(
    r"([\^\\']|\""
    r"\\.|"
    r"'([\^\\']|\""
    r"\\.|"
    r"'([\^\\']|\""
    r"\\.)*'")
```

```
_dquote3_finder = re.compile(
    r'([\^\\"]|'"
    r'\\.|"
    r'"([\^\\"]|'"
    r'\\.|"
    r'"([\^\\"]|'"
    r'\\.)*'")
```

```
# Reg. exps. to find the leftmost one-quoted string; use the
# "search" method; .span()[0] bounds the string found.
```

```
_dquote1_finder = re.compile(r'([\^"]|\\.)*')
```

```
_quotel_finder = re.compile(r"'^|\\.|'")

# _is_comment matches pure comment line.
_is_comment = re.compile(r"^[ \t]*#").match

# _is_blank matches empty line.
_is_blank = re.compile(r"^[ \t]*$").match

# find leftmost splat or quote.
_has_nightmare = re.compile(r"\"'#]").search

# _is_doc_candidate matches lines that start with a triple quote.
_is_doc_candidate = re.compile(r"^[ \t]*(\"'|\\\"\\'\\")")

del re

doc_type, comment_type, cod_type, blank_type = 'DOC', '###', 'COD', '---'

class Formatter:
    "Sort of a formatter class for filtering Python source code."

    def __init__(self, showLineNums=1):
        "Init."

        self.showLineNums = showLineNums
        self.showLineType = 0
        self.showLine = 0
        self.filterLineTypes = []

    def print_line(self, gnum, cnum, type, line):
        "Print a line, prefixed with some type and count information."

        if self.filterLineTypes and type in self.filterLineTypes:
            return

        if self.filterLineTypes:
            print '%s' % line,
            return

        if self.showLineNums and self.showLineType:
            if self.showLine and not self.filterLineTypes:
                print '%5d %5d %3s %s' % (gnum, cnum, type, line),

def crunch(getline, filename, mode):
    "Parse the given file."

    # for speed, give local names to compiled reps
    is_blank, is_comment, has_nightmare, is_doc_candidate = \
        _is_blank, _is_comment, _has_nightmare, _is_doc_candidate

    quote3_finder = { '": _dquote3_finder,
                      "'": _squote3_finder }
    quotel_finder = { '": _dquotel_finder,
                     "'": _squotel_finder }

    from re import sub

    num_code = num_comment = num_blank = num_doc = num_ignored = 0
    in_doc = in_triple_quote = lineno = 0
    while 1:
        # Eat one line.
        classified = 0
        lineno, line = lineno + 1, getline()
        if not line:
            break

        if in_triple_quote:
            if in_doc:
```

```

        num_doc = num_doc + 1
        mode.print_line(lineno, num_doc, doc_type, line)
    else:
        num_code = num_code + 1
        mode.print_line(lineno, num_code, cod_type, line)
    classified = 1
    m = in_triple_quote.match(line)
    if m == None:
        continue
    # Get rid of everything through the end of the triple.
    end = m.span()[1]
    line = line[end:]
    in_doc = in_triple_quote = 0

if is_blank(line):
    if not classified:
        num_blank = num_blank + 1
        mode.print_line(lineno, num_blank, blank_type, line)
    continue

if is_comment(line):
    if not classified:
        num_comment = num_comment + 1
        mode.print_line(lineno, num_comment, comment_type, line)
    continue

# Now we have a code line, a doc start line, or crap left
# over following the close of a multi-line triple quote; in
# (& only in) the last case, classified==1.
if not classified:
    if is_doc_candidate.match(line):
        num_doc = num_doc + 1
        in_doc = 1
        mode.print_line(lineno, num_doc, doc_type, line)
    else:
        num_code = num_code + 1
        mode.print_line(lineno, num_code, cod_type, line)

# The only reason to continue parsing is to make sure the
# start of a multi-line triple quote isn't missed.
while 1:
    m = has_nightmare(line)
    if not m:
        break
    else:
        i = m.span()[0]

    ch = line[i] # splat or quote
    if ch == '#':
        # Chop off comment; and there are no quotes
        # remaining because splat was leftmost.
        break
    # A quote is leftmost.
    elif ch*3 == line[i:i+3]:
        # at the start of a triple quote
        in_triple_quote = quote3_finder[ch]
        m = in_triple_quote.match(line, i+3)
        if m:
            # Remove the string & continue.
            end = m.span()[1]
            line = line[:i] + line[end:]
            in_doc = in_triple_quote = 0
        else:
            # Triple quote doesn't end on this line.
            break
    else:
        # At a single quote; remove the string & continue.
        prev_line = line[:i]
        line = sub(quotel_finder[ch], ' ', line, 1)
        # No more change detected, so be quiet or give up.

```

```
        if prev_line == line:
            # Let's be quiet and hope only one line is affected.
            line = ""
            # raise "ParseError", "Giving up at line %d" % lineno

answer = lineno-1, num_code, num_doc, num_comment, num_blank, filename
linenoSum = num_code + num_doc + num_comment + num_blank + 1 - num_ignored
if lineno != linenoSum:
    reason = ('internal inconsistency in counts', lineno, answer)
    raise SystemError, reason

return answer

def formatHeaderLine():
    format = '%8s%8s%8s%8s%8s %s'
    return format % ('lines', 'code', 'doc', 'comment', 'blank', 'file')

def formatResultLine(resTuple):
    # Print countings.
    format = '%8s%8s%8s%8s%8s %s'
    return format % resTuple

def collectFiles(listPatCwd, dirname, names):
    "Recursively add filenames matching a certain pattern to a list."

    list, pat, cwd = listPatCwd
    l = len(cwd)
    for name in names:
        p = os.path.join(dirname, name)
        if os.path.isfile(p) and fnmatch.fnmatch(name, pat):
            # Strip-off the current working directory from
            # the full path and replace it with a '.'.
            list.append('.' + p[l:])

def main():
    allTuples = []
    all = [0, 0, 0, 0, 0]

    verbose = 0
    recursive = 0
    is_filtered = 0
    is_first = 1
    m = Formatter()

    opts, args = getopt.getopt(sys.argv[1:], "vRF:")
    for o, a in opts:
        if o == '-v':
            m.showLineNums = 1
            m.showLineType = 1
            m.showLine = 1
        elif o == '-R':
            recursive = 1
        elif o == '-F':
            is_filtered = 1
            if string.find(a, ","):
                for t in string.split(a, ","):
                    m.filterLineTypes.append(t)
            else:
                m.filterLineTypes.append(a)

    # Handle stdin case.
    if len(args) == 0:
        resTuple = crunch(sys.stdin.readline, '<stdin>', m)
        if is_first and not is_filtered:
            print formatHeaderLine()
        if not is_filtered:
```

```
        print formatResultLine(resTuple)
    return

# Handle all other cases.

# Find files if we are in recursive mode.
if recursive:
    pat, cwd = args[0], os.getcwd()
    args = []
    os.path.walk(cwd, collectFiles, (args, pat, cwd))

for path in args:
    try:
        f = open(path, "r")
    except IOError, details:
        print "couldn't open %s: %s" % (path, details)
    else:
        try:
            resTuple = crunch(f.readline, path, m)
            allTuples.append(resTuple)
            if is_first and not is_filtered:
                print formatHeaderLine()
            if not is_filtered:
                print formatResultLine(resTuple)
        finally:
            is_first = 0
            f.close()

# Print total number in case we have more than one file.
if len(args) > 1:
    for t in allTuples:
        for i in (0, 1, 2, 3, 4):
            all[i] = all[i] + t[i]

    all.append("total")
    print formatResultLine(tuple(all))

if __name__ == '__main__':
    main()
```