

# Logging, scanning, and singleton objects for Python: `logscan.py` and `singleton.py`

John W. Shipman

2011-06-15 19:18

## Abstract

Describes Python classes that assist in scanning and reporting errors on data read from a stream.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to [tcc-doc@nmt.edu](mailto:tcc-doc@nmt.edu).

## Table of Contents

1. Introduction .....	2
2. Files available online .....	2
3. The singleton <code>Log</code> object .....	3
4. The <code>Scan</code> class: Managing progress through a stream .....	4
5. The code: Overview .....	8
6. Prologue to <code>logscan.py</code> .....	8
7. Imports .....	8
8. Manifest constants .....	8
8.1. <code>DEFAULT_PREFIX</code> .....	9
8.2. <code>ERROR_KIND</code> .....	9
8.3. <code>WARNING_KIND</code> .....	9
8.4. <code>WHITE_PATTERN</code> .....	9
8.5. <code>INT_PATTERN</code> .....	9
8.6. <code>FLOAT_PATTERN</code> .....	9
9. class <code>Log</code> .....	10
10. <code>Log.addLogFile()</code> : Log to another file .....	11
11. <code>Log.setPrefix()</code> : Change the prefix .....	12
12. <code>Log.msgKind()</code> : Send a message of a given kind .....	12
13. <code>Log.error()</code> : Send an error message .....	13
14. <code>Log.warning()</code> : Send a warning message .....	13
15. <code>Log.message()</code> : General message, not counted .....	13
16. <code>Log.write()</code> : Send a message to all current outputs .....	13
17. <code>Log.fatal()</code> : Write a message and stop .....	14
18. <code>Log.count()</code> : How many messages of a given kind? .....	14
19. <code>logtest</code> : Test driver for <code>Log</code> .....	14

<sup>1</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/>

<sup>2</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/logscan.pdf>

20. The <code>Scan</code> class .....	15
21. <code>Scan.__init__()</code> : The constructor .....	19
22. <code>Scan.__findInput()</code> : Find the input stream .....	20
23. <code>Scan.close()</code> : Close the stream .....	21
24. <code>Scan.atEndLine()</code> : Are we at the end of the current line? .....	21
25. <code>Scan.nextLine()</code> : Advance to the next line .....	21
26. <code>Scan.error()</code> : Issue an error message .....	22
27. <code>Scan.msgKind()</code> : General message writer .....	23
28. <code>Scan.move()</code> : Advance the scan position .....	24
29. <code>Scan.tab()</code> : Move to a specific position .....	24
30. <code>Scan.__effPos()</code> : Calculate an effective position .....	25
31. <code>Scan.isPos()</code> : Is the current position a given value? .....	25
32. <code>Scan.find()</code> : Search for a constant string .....	26
33. <code>Scan.upToRe()</code> : Search for a regular expression match .....	26
34. <code>Scan.deblankFile()</code> : Skip whitespace over multiple lines .....	27
35. <code>Scan.deblankLine()</code> : Skip whitespace on the current line .....	27
36. <code>Scan.match()</code> : Match a specific string .....	28
37. <code>Scan.matchArb()</code> : Case-insensitive match .....	28
38. <code>Scan.tabMatch()</code> : Advance if there is a match .....	29
39. <code>Scan.tabMatchArb()</code> : Case-insensitive match and move .....	29
40. <code>Scan.reMatch()</code> : Match a regular expression .....	30
41. <code>Scan.tabReMatch()</code> : Match and advance the position .....	30
42. <code>Scan.integer()</code> : Parse an integer .....	31
43. <code>Scan.fixed()</code> : Parse a float constant .....	32
44. <code>Scan.flatInt()</code> : Parse a fixed-size integer field .....	32
45. The singleton class: A classic design pattern .....	33
45.1. Code prologue for <code>singleton.py</code> .....	33
45.2. <code>class Singleton</code> .....	33
45.3. The <code>.__new__()</code> method .....	34
46. Test driver for <code>singleton</code> .....	34

## 1. Introduction

---

This document describes three related Python classes that the author has found useful for many kinds of input processing:

- The singleton `Log` object is used to centralize the processing of errors and error messages.
- The `Scan` class is used to coordinate the scanning of an input stream with the generation of error messages related to that stream.

In the first sections of this document, we will describe the interfaces to these classes. Later sections will walk you through the actual code.

## 2. Files available online

---

- `logscan.py`<sup>3</sup>: The module containing the `Log` and `Scan` classes.
- `singleton.py`<sup>4</sup>: The module containing the `singleton` class.

<sup>3</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/logscan.py>

<sup>4</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/singleton.py>

- `logtest`<sup>5</sup>: Test driver for the `Log` class.
- `singtest`<sup>6</sup>: Test driver for the `singleton` class.
- `logscan.xml`<sup>7</sup>: DocBook 4.3 source file for the current document.

### 3. The singleton `Log` object

---

In a lot of different applications written by the author, the same requirements crop up over and over again relative to error message handling:

- All error messages are always written to the standard error stream.
- A standardized prefix is attached to all error message lines, so they will stand out if mingled with other streams.
- If desired, all messages will also be written to one or more log files.
- The user can query to find out how many error messages have been issued up to the current time.
- The messages may be of multiple classes (error, warning, informational, and so on) and counts of each error class are maintained separately.

The `Log` class presented here centralizes these services. This class is an obvious application of the Singleton design pattern; see Section 45, “The `singleton` class: A classic design pattern” (p. 33). In practice, this means that the `Log()` constructor can be called any number of times from any number of locations in a program, but they will all use the same, single instance.

Here is the interface to the `Log` class.

#### **Log()**

Returns the singleton instance of `Log`. Initially, it will transmit messages only to the standard error stream.

#### **.addLogFile(*fileName*)**

Use this method to instruct the instance to send errors to a file whose name is *fileName*. If that file cannot be opened new for writing, it will raise an `IOError` exception.

If the file name is passed to the instance more than once, successive calls will be ignored.

#### **.setPrefix(*s*)**

Sets the string that is prefixed to all messages to *s*. The default value is `'*** '`.

#### **.msgKind (*kind*, \**L* )**

This method sends a message of a specific category. There are two predefined categories, whose values (as strings) are given by the constants `ERROR_KIND` (currently `"Error"`) and `WARNING_KIND` (currently `"Warning"`). You may define additional categories by passing the category name as the `msgKind` argument.

The remaining arguments must also be strings, which are all concatenated together and written to all the current message destinations, with each line preceded by the current prefix, and the contents of the first line also preceded by the category name and a colon.

Here's an example of the call. Assume that `nGooFs` is 3 and the standard error prefix is still in force.

---

<sup>5</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/logtest>

<sup>6</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/singtest>

<sup>7</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/logscan/logscan.xml>

```
Log().msgKind('Notice', 'There have been ', str(nGoofs),
' goofs so far.\nThese goofs were not fatal.' )
```

Here's the output that call will send to the current set of destinations:

```
*** Notice: There have been 3 goofs so far.
*** These goofs were not fatal.
```

#### **.error(\*L)**

Sends an error message. Same as `.msgKind(ERROR_KIND, *L)`.

#### **.warning(\*L)**

Sends a warning message. Same as `.msgKind(WARNING_KIND, *L)`.

#### **.message ( \*L )**

Sends a message to the current destinations. The arguments are one or more strings, which are concatenated to form the message. The current prefix will be prepended to each line of the message.

#### **.write ( \*L )**

Like `.message ( )`, but does not prepend the current prefix.

#### **.fatal ( \*L )**

To write a final message *and then terminate execution of the program*, call this method, and pass it one or more strings, which will be concatenated to form the message.

#### **.count ( kind=None )**

Returns the number of messages of the given `kind` since the first instantiation of this class. If no `kind` argument is passed, `ERROR_KIND` is the default.

## 4. The Scan class: Managing progress through a stream

---

The `Scan` class was invented to manage the scanning of a file, especially when error messages may be written that are related to locations in the file. It is intended for writing small compilers and other applications that syntax-check files. All error and other message logging is handled using the `Log` singleton described in Section 3, “The singleton `Log` object” (p. 3).

This method of stream processing assumes:

- that the file is structured into lines separated by ASCII newline characters; and
- that the caller will never want to back up to a previous line, though backing up within a line is allowed.

These constraints guarantee that minimal amounts of storage will be taken up even while reading quite large input files, so long as individual lines are not too large.

Each `Scan` instance keeps track of the current position in the current line. A number of different methods allow you to move the scan position forward (and backward, within the line, if you like), and to advance to the next line (if there is one).

The `Scan` class also supports an optional line-comment character. For example, you might ask for your `Scan` instance to ignore a percent sign (%) and everything after it on that line.

Often it is useful to identify the site of an error with more information than just the file name and line number. The `Scan` object also allows you to provide a callback procedure that can furnish a string that identifies where you are in the file. For example, suppose you are writing a compiler for a language that divides code into modules. The callback procedure might identify which module you are currently in.

This Python class is based on an earlier version implemented in Icon, which has a large and well-designed set of scanning primitives. Ralph Griswold's book, *The Icon Programming Language*, explains these scanning primitives, but you won't need to know Icon to use the Python `Scan` object.

Here is the interface to the `Scan` class.

### **Scan(*inFile*, *commentPrefix*=None, *callback*=None)**

The required *inFile* argument specifies the stream to be read. The constructor attempts to read the first line; if the stream is empty, it will set the `.atEndFile` attribute.

- If *inFile* is a string, `Scan` will attempt to open a file by that name. If the open fails, the constructor will raise an `IOError` exception.
- *inFile* may also be a file or file-like object that provides `.readline()` and `.close()` methods.

If a `commentPrefix` string is specified, all input lines that start with that string will be treated as comments. The default behavior is not to consider anything a comment.

If you want to provide additional information in some error messages, pass a function reference as the `callback` argument. Whenever an error message is issued through the `Scan` instance, this procedure will be called, with the `Scan` instance as its sole argument, and the result of this call (which should be a string) will be appended to the error message. This is provided so that the application can identify the point of the error relative to the file's logical structure, not just the current line's contents and its physical position in the file.

#### **.atEndFile**

This read-only attribute is a `bool`, initially false, and set to `True` when the end of the stream has been reached.

#### **.line**

The contents of the current line, as a string, without any trailing line termination. It may be empty. If there is a `commentPrefix` in force, the comment part of the line is removed. Read-only.

#### **.rawLine**

Same as the `.line` attribute except that the comment is not removed if there is one. Read-only.

#### **.lineNo**

The current line number, counting from 1. Read-only.

#### **.pos**

Index of the current scan position within the current line, in `range(len(line))`. Read-only.

#### **.close()**

Call this method when stream scanning is complete.

#### **.atEndLine()**

A predicate that returns `True` if the current scan position is at the end of the current line, `False` otherwise.

#### **.nextLine()**

No matter where the scan position is on the current line, try to move to the start of the next line. If there are no lines remaining, leave the position at the end of the file. You can test to see if you are at the end of the file by using the `.atEndFile()` predicate described above. Returns `True` if the current line was not the last, `False` otherwise.

#### **.error(\*L)**

Writes an error message.

- If there have been no previous messages issued for the current line, the line's contents are sent to `Log()`, followed by the result returned by the callback procedure (if there is one).

- A carat (^) is displayed under the current scan position.
- The arguments are the error message, as one or more strings that are concatenated to form the message. The complete message is then sent to `Log().error()`.

#### **.syntax(\*L)**

Works the same as the `.error()` method, but after transmitting the error message, it raises a `SyntaxError` exception.

#### **.warning(\*L)**

Works the same as `.error()`, but the message is sent to the `Log().warning()` method.

#### **.msgKind(kind, \*L)**

The message is built up as in the `.error()` method, but the result is sent to `Log().msgKind(kind)`.

#### **.move(n)**

If at least  $n$  characters remain on the current line, advances the position that far and returns the characters between the current position and the new position. If the current line has fewer than  $n$  characters remaining, raises `IndexError`.

#### **.tab(p)**

Try to move the position within the current line to  $p$ . If successful, it returns the string between the current position and  $p$ .

If  $p$  is nonnegative, it is treated as the normal Python string index—0 for the first character, 1 for the second, and so on.

You may also describe positions relative to the end of the string, but these *do not follow the standard Python behavior for negative indices*. Position -1 is at the end of the string; -2 is before the last character on the line; and so on.

If position  $p$  is within the line, the position is set as requested. If it is out of range for the current line length, this method will raise `IndexError`.

#### **.isPos(p)**

A predicate to test whether the position in the current line is equivalent to  $p$ , where the values of  $p$  are as described above for the `.tab()` method.

For example, to test to see if there is exactly one character remaining on the line in scan object `s`, you could use the expression `s.isPos(-2)`, which would return `True` if you are positioned on the last character, `False` otherwise.

#### **.find(s)**

This method searches the remainder of the current line for a string `s`. If the string is found, this method returns the index on the current line (counting from 0) where the first match begins. If string `s` does not occur in the remaining part of the current line, the method returns `None`.

#### **.upToRe(r)**

This method is used to look for a specific pattern on the current line, at or after the current position. The argument must be a regular expression (as a string) or a compiled regular expression.

If the pattern is found, this method returns the index on the current line where the first match starts. If the pattern is not found, it returns `None`.

#### **.deblankFile()**

Tries to advance to the next non-whitespace character in the file, if any. This method may move beyond the current line, unlike most of the scanning methods. If there are no characters remaining in the file, or the next character is not a whitespace character, it does nothing.

### **.deblankLine()**

If there is anything left on the current line, and the character at the current scan position is a whitespace character, the position is advanced to the next non-whitespace character, but not past the end of the current line. Otherwise it does nothing.

### **.match(s)**

The argument *s* must be a nonempty string. If the current line at the current position starts with *s*, the method returns the position just after the match; otherwise it returns `None`.

### **.matchArb(s)**

Like `.match()`, but ignores case—that is, it treats all letters in both *s* and the current line as if they were uppercased.

### **.tabMatch(s)**

Like `.match()`, but if *s* matches, the current position is moved to a point just after the match, and the matching contents are returned.

### **.tabMatchArb(s)**

Like `.tabMatch()`, but ignores case.

### **.reMatch(r)**

The argument *r* is a regular expression (in string or compiled RE form). If the current line beginning at the current position matches *r*, the method returns the `MatchObject` instance; otherwise it returns `None`.

### **.tabReMatch(r)**

If the string or compiled regular expression *r* matches the current line at the current position, returns a `MatchObject` instance and advances the position to a point just after the matching content; otherwise it returns `None`.

### **.integer(maxLen=None)**

If the current line at the current position starts with one or more digits, optionally preceded by + or -, the position is advanced past those characters, and the value is returned *as an int*; otherwise it returns `None`.

If you wish to limit the length of the matched number, pass the maximum number of digits in as an argument.

### **.fixed()**

If the current line at the current position starts with a float (that is, one or more digits and at most one decimal point), optionally preceded by a sign, this method advances the position to a point just after the float, and returns the matching part *as a float*; otherwise it returns `None`.

## **Warning**

Note that this does not allow fixed-point numbers to start with a decimal point, so the string `".1"` would not be considered valid. The author feels that fixed-point constants less than one should always have a leading zero, because in a number like `".1"` it is too easy to miss the decimal point or ignore it as a flyspeck on the monitor.

### **.flatInt ( n )**

If the current line at the current position starts with an integer in a fixed-size field of length *n*, the method advances the position to a point just after that field and returns the integer as type `int`; otherwise it returns `None`.

## 5. The code: Overview

---

The balance of this document contains the actual code for the interfaces described above, using the technique of *lightweight literate programming*<sup>8</sup>.

All this code was developed using the *Cleanroom software development methodology*<sup>9</sup>. Comments in [ square brackets ] are Cleanroom intended functions, a notation especially good for describing interfaces exactly.

## 6. Prologue to `logscan.py`

---

The `logscan.py` module starts with a comment directing the reader back to this documentation.

`logscan.py`

```
'''logscan.py: Log and Scan objects for Python.  
  
Do not edit this file. It is extracted automatically from the  
documentation:  
    http://www.nmt.edu/tcc/help/lang/python/examples/logscan/  
...'''
```

## 7. Imports

---

We need the standard Python `sys` module to access the standard error stream, and the `re` module for regular expression operations.

`logscan.py`

```
#===== # Imports #-----  
  
import sys  
import re
```

The `Log` class inherits from the `singleton` class described in Section 45, “The singleton class: A classic design pattern” (p. 33).

`logscan.py`

```
from singleton import *
```

## 8. Manifest constants

---

These names are constants used inside and outside the module.

`logscan.py`

```
#===== # Manifest constants #-----
```

---

<sup>8</sup> <http://www.nmt.edu/~shipman/soft/litprog/>

<sup>9</sup> <http://www.nmt.edu/~shipman/soft/clean/>

## 8.1. DEFAULT\_PREFIX

Default prefix for the Log class.

logscan.py

```
DEFAULT_PREFIX = "*** "
```

## 8.2. ERROR\_KIND

The code for error messages.

logscan.py

```
ERROR_KIND = "Error"
```

## 8.3. WARNING\_KIND

The code for warning messages.

logscan.py

```
WARNING_KIND = "Warning"
```

## 8.4. WHITE\_PATTERN

A compiled regular expression that matches zero or more whitespace characters.

logscan.py

```
WHITE_PATTERN = re.compile ( r'\s*' )
```

## 8.5. INT\_PATTERN

A regular expression that matches valid integers.

logscan.py

```
INT_REGEX = (  
    r'\s*'          # Ignore leading whitespace  
    r'([\-+]?[s*])' # Optional sign and following whitespace  
    r'\d+'         # One or more digits left of decimal  
INT_PATTERN = re.compile ( INT_REGEX )
```

## 8.6. FLOAT\_PATTERN

A regular expression that matches valid floats.

logscan.py

```
FLOAT_REGEX = (INT_REGEX +  
    r'('          # Start optional fraction group  
    r'\.'        # Decimal point  
    r'\d*'       # Zero or more digits after it  
    r')?'       # Whole group is optional  
FLOAT_PATTERN = re.compile ( FLOAT_REGEX )
```

## 9. class Log

The intended functions below are the formal descriptions of the interface. They should match the informal descriptions given in Section 3, "The singleton Log object" (p. 3). First we define a verification function named `log-outputs` to describe the current set of places where messages will be sent.

logscan.py

```
#####
# Verification functions
#-----
# log-outputs(Log) ==
# (sys.stderr) + (all log files of Log)
#-----

# - - - - - c l a s s   L o g

class Log(Singleton):
    '''Error logging object.

    Exports:
    Log():
        [ if Log has been instantiated before ->
          return that instance
          else ->
            return a new instance logging only to sys.stderr,
            with prefix DEFAULT_PREFIX and no error counts ]
    .addLogFile ( fileName ):
        [ fileName is a string ->
          if fileName has been passed to this method before ->
            I
          else if fileName can be opened new for writing ->
            self := self with that file so opened and added
            to log-outputs(self) ]
          else -> raise IOError ]
    .setPrefix ( s ):
        [ s is a string ->
          self := self with the current prefix changed to s ]
    .msgKind ( kind, *L ):
        [ if (kind) is a key in self.__kindCounts ->
          self.__kindCounts[kind] += 1
          else ->
            self.__kindCounts[kind] := 1
          In any case ->
            log-outputs(self) := lines made from the
            concatenation of *L, broken on newlines, with each
            line preceded by the current prefix, and the text
            of the first line preceded by (kind) ]
    .error ( *L ):
        [ equivalent to self.msgKind ( ERROR_KIND, *L ) ]
    .warning ( *L ):
        [ equivalent to self.msgKind ( WARNING_KIND, *L ) ]
    .message ( *L ):
```

```

    [ L is a list of strings ->
      log-outputs(self) := lines made from the concatenation
                          of L, broken on newlines, with each line preceded
                          by the current prefix ]
.write ( *L ):
    [ L is a list of strings ->
      log-outputs(self) := the concatenation of L ]
.fatal ( *L ):
    [ L is a list of strings ->
      log-outputs(self) := lines made from the concatenation
                          of L, broken on newlines, with each line preceded
                          by the current prefix
      stop execution ]
.count ( kind ):
    [ kind is a string ->
      if kind is a key in self.__kindCounts ->
        return self.__kindCounts[kind]
      else -> return 0 ]

```

Here are the internal state attributes of the class.

logscan.py

```

State/Invariants:
.___prefix:
    [ the current prefix string ]
.___logMap:
    [ a dictionary whose keys are the names of log files
      requested by .addLogFile, and each related value is
      that file, opened for writing ]
.___kindCounts:
    [ a dictionary whose keys are the unique (kind) arguments
      passed to .msgKind(), and each related value is the
      count of messages of that kind ]
...

```

Because this is a singleton, the constructor may be called many times on the same instance. Therefore, we'll just establish the class invariants as class variables. This means we don't need to define a constructor.

logscan.py

```

__prefix = DEFAULT_PREFIX
__logMap = {'': sys.stderr}
__kindCounts = {}

```

## 10. Log.addLogFile(): Log to another file

logscan.py

```

# - - -   L o g . a d d L o g F i l e

def addLogFile ( self, logFileName ):
    '''Add another logging destination.
    ...

```

If `logFileName` is a name we've seen before, do nothing—we don't want to overwrite the existing file. Otherwise, try to open it, then add it to `self.__logMap`.

```

#-- 1 --
if logFileName in self.__logMap:
    return

#-- 2 --
# [ if logFileName can be opened new for writing ->
#     self.__logMap[logFileName] := that file, so opened
#     else -> raise IOError ]
self.__logMap[logFileName] = file ( logFileName, 'w' )

```

## 11. Log.setPrefix(): Change the prefix

```

# - - -   L o g . s e t P r e f i x

def setPrefix ( self, s ):
    '''Change the current message prefix.
    ...
    self.__prefix = s

```

## 12. Log.msgKind(): Send a message of a given kind

```

# - - -   L o g . m s g K i n d

def msgKind ( self, kind, *L ):
    '''Send a message and tally one message of this (kind).
    ...
    #-- 1 --
    # [ log-outputs(self) += lines made from (kind)+(the
    #     concatenation of *L), broken on newlines, with
    #     each line preceded by the current prefix ]
    self.message ( "%s: %s" %
                   (kind, ''.join(L)) )

    #-- 2 --
    # [ if kind is a key in self.__kindCounts ->
    #     self.__kindCounts[kind] += 1
    #     else ->
    #     self.__kindCounts[kind] := 1 ]
    try:
        self.__kindCounts[kind] += 1
    except KeyError:
        self.__kindCounts[kind] = 1

```

## 13. Log.error(): Send an error message

---

logscan.py

```
# - - -   L o g . e r r o r

def error ( self, *L ):
    '''Send a message of kind ERROR_KIND.
    ...

    self.msgKind ( ERROR_KIND, *L )
```

## 14. Log.warning(): Send a warning message

---

logscan.py

```
# - - -   L o g . w a r n i n g

def warning ( self, *L ):
    '''Send a message of kind WARNING_KIND.
    ...

    self.msgKind ( WARNING_KIND, *L )
```

## 15. Log.message(): General message, not counted

---

logscan.py

```
# - - -   L o g . m e s s a g e

def message ( self, *L ):
    '''Send a message, but don't increment any counts.
    ...
```

The big job here is to break up the message on newlines, and attach the current prefix to each line.

logscan.py

```
#-- 1 --
for line in (''.join(L)).split('\n'):
    #-- 1 body --
    # [ log-outputs(self) += self.__prefix + line ]
    self.write ( self.__prefix + line )
```

## 16. Log.write(): Send a message to all current outputs

---

logscan.py

```
# - - -   L o g . w r i t e

def write ( self, *L ):
    '''Write a message to log-outputs(self).
    ...

    #-- 1 --
    text = ''.join(L)

    #-- 2 --
```

```
for outFile in self.__logMap.values():
    print >>outFile, text
```

## 17. Log . fatal ( ) : Write a message and stop

logscan.py

```
# - - - L o g . f a t a l

def fatal ( self, *L ):
    '''Write a message and terminate execution.
    ...

    self.message ( ''.join(L) )
    raise SystemExit
```

## 18. Log . count ( ) : How many messages of a given kind?

logscan.py

```
# - - - L o g . c o u n t

def count ( self, kind=None ):
    '''Return the number of messages of a given kind.
    ...
```

We need to return zero if there have never been any messages of this kind. Otherwise, the count is in self.\_\_kindCounts. The default kind is ERROR\_KIND.

logscan.py

```
#-- 1 --
if kind is None:
    kind = ERROR_KIND

#-- 2 --
try:
    return self.__kindCounts[kind]
except KeyError:
    return 0
```

## 19. logtest: Test driver for Log

Here is a small test script to exercise the functions of the Log object.

logtest

```
#!/usr/bin/env python
#=====
# Test driver for the Log class.
#
# Do not edit this file. It is extracted automatically from the
# documentation:
# http://www.nmt.edu/tcc/help/lang/python/examples/logscan/
#-----
from logscan import *
```

```

# - - -   m a i n

def main():
    '''Test the Log class.
    ...
    NOTE_KIND = "Note"

    Log().message("Sent only to ", "stderr.")
    Log().addLogFile('a.log')
    Log().msgKind(NOTE_KIND, "This should be sent to a.log\n"
                  "but not to b.log.")
    Log().setPrefix('%%% ' )
    Log().msgKind(NOTE_KIND, "This should be prefixed with percents\n"
                  "and not stars.")
    Log().addLogFile('b.log')
    Log().warning("This should appear in both a.log and b.log.")
    Log().error("This is an error message.")
    print "Count of errors, should be 1:", Log().count()
    print "Count of warnings, should be 1:", Log().count(WARNING_KIND)
    print "Count of notes, should be 2:", Log().count(NOTE_KIND)
    print "And now, the fatal test:"
    Log().fatal("This is the fatal message.")
    print "This message had better not appear!"

#=====
# Epilogue
#-----
if __name__ == '__main__':
    main()

```

## 20. The Scan class

The intended functions below are the formal descriptions of the interface. They should match the informal descriptions given in Section 4, “The Scan class: Managing progress through a stream” (p. 4).

Before the actual class interface definition, we define one verification function that describes the behavior of position arguments to methods such as `.tab()`.

logscan.py

```

#=====
# Verification functions
#-----
# effective-pos(p) ==
#   if p is nonnegative ->
#       p
#   else ->
#       (length of the current line) + 1 + p
#-----

# - - - - -   c l a s s   S c a n

```

```

class Scan(object):
    '''Stream scanning class.

    Exports:
    Scan(inFile, commentPrefix=None, callback=None):
        [ (inFile is a string or file-like object) and
          (commentPrefix is a string or None) and
          (callback is a function or None) ->
            if (fileName names a readable file) ->
                return a new Scan object with (commentPrefix)
                as the comment character (if given) and
                using (callback) as the callback procedure
                (if given) and positioned at the beginning
                of the first line (if there is one)
            else if (fileName is a readable file handle) ->
                return a new Scan object with (commentPrefix)
                as the comment character (if given) and
                using (callback) as the callback procedure
                (if given), reading from fileName at its
                current position
            else ->
                raise IOError ]
    .atEndFile: # Read-only
        [ if self is positioned at the end of the stream ->
          True
          else -> False ]
    .line: # Read-only
        [ if self is not at the end of the stream ->
          the current line, with line terminator and
          comment (if any) removed
          else -> (undefined) ]
    .rawLine: # Read-only
        [ if self is not at the end of the stream ->
          the current line, with line terminator removed
          else -> (undefined) ]
    .lineNo: # Read-only
        [ the line number of the current line in self,
          counting from 1 ]
    .pos: # Read-only
        [ if self is not at the end of the stream ->
          the position within the current line, counting
          from 0
          else -> (undefined) ]
    .close(): [ self := self, closed ]
    .atEndLine():
        [ if self is at the end of the stream or the end of
          the current line ->
          return True
          else -> return False ]
    .nextLine():
        [ if self is at the end of the stream ->
          return False

```

```

        else ->
            self := self advanced to the beginning of the
                    next line ]
.error(*L):
    [ L is a list of strings ->
      if any message has been issued for the current line ->
          Log() += an error message showing the current
                  line and position of self, with the concatenated
                  elements of L used as the message
      else ->
          Log() += (current raw line) + (an error
                  message showing the current line and position
                  of self, with the concatenated elements of L
                  used as the message )
.syntax(*L):
    [ like .error() but raises SyntaxError after
      transmitting the error message ]
.warning(*L):
    [ like .error() but issues a warning rather than an
      error message ]
.msgKind(kind, *L):
    [ (kind is a string) and
      (L is a list of strings) ->
      like .error(), but issues a message of kind (kind) ]
.message(*L):
    [ like .error() but uses Log().message() ]
.write(*L):
    [ like .error() but uses Log().write() ]
.move(n):
    [ n is a nonnegative int ->
      if there are at least n characters remaining on the
      current line ->
          self := self with the position advanced by n
      else -> raise IndexError ]
.tab(p):
    [ if effective-pos(p) is within the current line ->
      self := self with the current position moved to
              effective-pos(p)
      else -> raise IndexError ]
.isPos(p):
    [ if effective-pos(p) is the current position in the
      current line ->
      return True
      else -> return False ]
.find(s):
    [ s is a string ->
      if any part of the remainder of the current line
      matches s ->
          return the position on the current line where the
          first such match begins
      else -> return None ]
.upToRe(r):
    [ r is a regular expression in string or compiled form ->

```

```

        if any part of the remainder of the current line
        matches r ->
            return the position on the current line where the
            first such match begins
        else -> return None ]
.deblankFile():
    [ self := self advanced past any leading whitespace,
      over any number of lines ]
.deblankLine():
    [ self := self advanced past any leading whitespace,
      but not past end of line ]
.match(s):
    [ s is a string ->
      if the current position starts with s ->
          return the position just after the match
      else -> return None ]
.matchArb(s):
    [ s is a string ->
      if the current position starts with s, case-insensitive ->
          return the position just after the match
      else -> return None ]
.tabMatch(s):
    [ s is a string ->
      if the current line starts with s ->
          self := self advanced past the match
          return s
      else -> return None ]
.tabMatchArb(s):
    [ s is a string ->
      if the current line starts with s, case-insensitive ->
          self := self advanced past the match
          return s
      else -> return None ]
.reMatch(r):
    [ r is a regular expression as a string or compiled ->
      if r matches at the current position on the current
      line ->
          return a MatchObject representing the match
      else -> return None ]
.tabReMatch(r):
    [ r is a regular expression as a string or compiled ->
      if r matches at the current position on the current
      line ->
          self := self advanced past the matching part
          return a MatchObject representing the match
      else -> return None ]
.integer(maxLen=None):
    [ maxLen is an int, defaulting to 2**31-1 ->
      if the current line at the current position starts
      with one or more digits, preceded by an optional
      "+" or "-" ->
          self := self advanced past all that
          return all that as an int

```

```

        else -> return None ]
.fixed():
    [ if the current line at the current position starts
      with one or more digits, optionally preceded by
      "+" or "-", and containing at most one "." ->
      self := self advanced past all that
      return all that as a float
      else -> return None ]
.flatInt(n):
    [ n is a positive integer ->
      if the current line at the current position starts
      with an integer right-justified in a field of size n ->
      self := self advanced by n
      return that integer as type int
      else -> return None ]

```

The above is the exported interface. Next come the internal state variables.

logscan.py

```

State/Invariants:
.fileName:
    [ if the constructor was called with a string ->
      that string
      else -> None ]
.file:
    [ if self is closed -> None
      else ->
        readable file handle for the input stream ]
.commentPrefix: [ as passed to the constructor ]
.callback:       [ as passed to the constructor ]
.__echoed:
    [ if any messages have been issued during the current
      line ->
      True
      else -> False ]
...

```

## 21. Scan.\_\_init\_\_(): The constructor

logscan.py

```

# - - -   S c a n . _ _ i n i t _ _

def __init__( self, inFile, commentPrefix=None, callback=None ):
    '''Constructor
    ...

    #-- 1 --
    self.commentPrefix = commentPrefix
    self.callback = callback
    self.atEndFile = False
    self.line = ""
    self.rawLine = ""
    self.lineNo = 0
    self.pos = 0

```

```

self.__echoed = False

#-- 2 --
# [ if inFile is a string ->
#     if inFile can be opened for reading ->
#         self.fileName := inFile
#         self.file := that file, so opened
#     else -> raise IOError
#         self.fileName = None
#         self.file := inFile ]
self.__findInput ( inFile )

#-- 3 --
# [ if self.file is nonempty ->
#     self.file := self.file advanced past the first line
#     self := self with the first line positioned at
#                 the first character
#     else ->
#         self := self at end of file ]
self.nextLine()

```

## 22. Scan.\_\_findInput(): Find the input stream

logscan.py

```

# - - - S c a n . _ _ f i n d I n p u t

def __findInput ( self, inFile ):
    '''Take care of opening the input if necessary.

    [ if inFile is a string ->
        if inFile can be opened for reading ->
            self.fileName := inFile
            self.file := that file, so opened
        else -> raise IOError
            self.fileName = None
            self.file := inFile ]
    ...

#-- 1 --
# [ if inFile is not a string ->
#     self.fileName := None
#     self.file := inFile
#     return
#     else ->
#         self.fileName := inFile ]
if not isinstance(inFile, basestring):
    self.fileName = None
    self.file = inFile
    return
self.fileName = inFile

#-- 2 --
# [ if inFile can be opened for reading ->

```

```
# self.file := that file, so opened
# else -> raise IOError ]
self.file = open ( inFile )
```

## 23. Scan.close(): Close the stream

logscan.py

```
# - - - S c a n . c l o s e

def close ( self ):
    '''Close the input stream.
    ...
    if self.file:
        self.file.close()
        self.file = None
```

## 24. Scan.atEndLine(): Are we at the end of the current line?

logscan.py

```
# - - - S c a n . a t E n d L i n e

def atEndLine ( self ):
    '''Predicate: is the scan position at end of line?
    ...
```

If the file is closed or at end of file, that's considered end of line. Otherwise, it is end of line if the current position is equal to the length of the line.

logscan.py

```
return ( (self.file is None) or
         (self.atEndFile) or
         (self.pos == len(self.line)) )
```

## 25. Scan.nextLine(): Advance to the next line

logscan.py

```
# - - - S c a n . n e x t L i n e

def nextLine ( self ):
    '''Advance to the next line, if there is one.
    ...
    #-- 1 --
    if self.atEndFile:
        return False
    else:
        self.pos = 0
```

Next we use the file's `.readline()` method to try to read the next line. This method returns zero at the end of the stream.

```

#-- 2 --
# [ if self.file has at least one line remaining ->
#     self.file := self.file advanced past the next line
#     self.rawLine := next line from self.file
# else ->
#     self.rawLine := '' ]
self.rawLine = self.file.readline()

#-- 3 --
if len(self.rawLine) == 0:
    self.atEndFile = 1
    return False
else:
    self.lineNo += 1
    self.__echoed = False

```

If there is a trailing newline, we remove it. Then, we set `self.line` to `self.rawLine`, with the comment removed if there is one.

```

#-- 4 --
# [ if self.rawLine ends with a newline ->
#     self.rawLine := self.rawLine without that newline
# else -> I ]
if ( ( len(self.rawLine) > 0 ) and
      ( self.rawLine[-1] == '\n' ) ):
    self.rawLine = self.rawLine[:-1]

#-- 5 --
# [ if (self.commentPrefix is not None) and
#     (self.commentPrefix matches anywhere on self.line) ->
#     self.line := self.line truncated at the start of
#                 the first match
# else ->
#     self.line := self.rawLine ]
self.line = self.rawLine
if self.commentPrefix:
    commentPos = self.rawLine.find(self.commentPrefix)
    if commentPos >= 0:
        self.line = self.rawLine[:commentPos]

#-- 6 --
return True

```

## 26. Scan.error(): Issue an error message

These methods all pass the messages through to the `Log()` singleton, but they also take care of echoing the current position, calling the callback if there is one, and also echoing the current line when this is the first message for the current line.

```

# - - -   S c a n . e r r o r
# - - -   S c a n . s y n t a x
# - - -   S c a n . w a r n i n g

def error ( self, *L ):
    '''Send an error message to Log().
    ...

    self.msgKind ( ERROR_KIND, *L )

def syntax ( self, *L ):
    '''Send an error message and raise SyntaxError.
    ...

    self.error ( *L )
    raise SyntaxError ( ''.join(L) )

def warning ( self, *L ):
    '''Send a warning to Log().
    ...

    self.msgKind ( WARNING_KIND, *L )

```

## 27. Scan.msgKind(): General message writer

```

# - - -   S c a n . m s g K i n d

def msgKind ( self, kind, *L ):
    '''General messages to Log().
    ...

```

If this is the first message for the current line, echo the line's content. If there is a callback, it is activated only when the line is first echoed.

```

#-- 1 --
# [ if self.__echoed is False ->
#     self.__echoed := True
#     Log() := (description of the current file position) +
#             (result of self.callback, if any) + (self.rawLine)
# else -> I ]
if not self.__echoed:
    if self.fileName:
        where = ( "File '%s', line %d" %
                  (self.fileName, self.lineNo) )
    else:
        where = ( "Line %d" % self.lineNo )

    if self.callback:
        where = "%s [%s]" % (where, self.callback(self))

    Log().write ( "\n--- ", where, "\n", self.rawLine )
    self.__echoed = True

```

```

#-- 2 --
# [ Log() += (a line pointing to self.pos) +
#       (concatenation of elements of L) ]
Log().write ( " " * self.pos, "^" )
Log().msgKind ( kind, *L )

```

## 28. Scan.move(): Advance the scan position

logscan.py

```

# - - - S c a n . m o v e

def move ( self, n ):
    '''Move the scan position by n characters.
    ...

```

The value of `n` may be negative. First, check that the new position is somewhere within the current line. The new position may be at the end of the line, so the case `(self.pos+n == len(self.line))` is valid.

logscan.py

```

#-- 1 --
# [ if self.pos + n is within the current line ->
#     newPos := self.pos + n
#     else -> raise IndexError ]
newPos = self.pos + n
if not (0 <= newPos <= len(self.line)):
    raise IndexError("Scan.move: new position out of range.")

```

Whether the move was backwards or forwards, we return the string between the old and new positions.

logscan.py

```

#-- 2 --
loPos = min(self.pos, newPos)
hiPos = max(self.pos, newPos)
self.pos = newPos

#-- 3 --
return self.line[loPos:hiPos]

```

## 29. Scan.tab(): Move to a specific position

logscan.py

```

# - - - S c a n . t a b

def tab ( self, p ):
    '''Move to position p in the current line.
    ...

#-- 1 --
# [ newPos := effective-pos(p) ]
newPos = self.__effPos ( p )

```

The rest of the logic is similar to Section 28, “Scan.move(): Advance the scan position” (p. 24), as if we were moving the position by an amount `newPos - self.pos`.

```

#-- 2 --
# [ if newPos is within the current line ->
#     self := self positioned at newPos in the current line
#     return text between self's current position and newPos
#     else -> raise IndexError ]
return self.move ( newPos - self.pos )

```

### 30. Scan.\_\_effPos(): Calculate an effective position

```

# - - -   S c a n . _ _ e f f P o s

def __effPos ( self, p ):
    '''Return the equivalent position for position p.

    [ p is an int ->
      return effective-pos(p) ]
    ...

```

For the definition of the effective-pos verification function, see Section 20, “The Scan class” (p. 15).

```

if p < 0:
    return len(self.line) + 1 + p
else:
    return p

```

### 31. Scan.isPos(): Is the current position a given value?

```

# - - -   S c a n . i s P o s

def isPos ( self, p ):
    '''Is the current line at position p?
    ...

```

See Section 30, “Scan.\_\_effPos(): Calculate an effective position” (p. 25) for the routine that converts negative positions to positive values.

```

#-- 1 --
# [ newP := effective-pos(p) ]
newP = self.__effPos ( p )

#-- 2 --
return (self.pos == newP)

```

## 32. Scan.find(): Search for a constant string

logscan.py

```
# - - -   S c a n . f i n d

def find ( self, s ):
    '''Does string s occur at or after the current position?
    ...
```

We use the standard `str.find` method to do the searching. This method returns -1 if the string is not found.

logscan.py

```
#-- 1 --
# [ if s is found at or after self.line[self.pos:] ->
#     result := position of the first match relative
#             to self.line
# else ->
#     result := -1 ]
result = self.line.find ( s, self.pos )

#-- 2 --
if result < 0:
    return None
else:
    return result
```

## 33. Scan.upToRe(): Search for a regular expression match

logscan.py

```
# - - -   S c a n . u p T o R e

def upToRe ( self, r ):
    '''Is there a match for regex (r) on this line?
    ...
```

First we use `re.search()` to see if there is a match for `r` at or after the current position on the current line. The `re.search()` method takes a third argument specifying the starting position.

logscan.py

```
#-- 1 --
# [ if (r is a string) and (regex (r) matches within
#     self.line[self.pos:]) ->
#     m := a MatchObject representing that match
# else if compiled regex (r) matches within
#     self.line[self.pos:]) ->
#     m := a MatchObject representing that match
# else -> return None ]
if isinstance(r, basestring):
    m = re.search ( r, self.line, self.pos )
else:
    m = r.search ( self.line, self.pos )
if m is None:
    return None
```

Since `m` is now a `MatchObject` describing the matched portion, we can use `m.start()` to find the position of the start of the match.

logscan.py

```
#-- 2 --  
return m.start()
```

## 34. `Scan.deblankFile()`: Skip whitespace over multiple lines

logscan.py

```
# - - -   S c a n . d e b l a n k F i l e  
  
def deblankFile ( self ):  
    '''Skip whitespace until end of file  
    ...  
#-- 1 --  
while not self.atEndFile:  
    #-- 1 body --  
    # [ if there is a nonblank character at on the current  
    #   line at or after the current position ->  
    #     self := self advanced to that character  
    #     return  
    #   else ->  
    #     self := self advanced to the start of the  
    #           next line ]  
  
    #-- 1.1 --  
    # [ self := self advanced past all whitespace on the  
    #     current line at or after the current position ]  
    self.deblankLine()  
  
    #-- 1.2 --  
    # [ if self is at end of line ->  
    #     self := self advanced to the next line or to  
    #           end of file, whichever comes first  
    #   else -> return ]  
    if self.atEndLine():  
        self.nextLine()  
    else:  
        return
```

## 35. `Scan.deblankLine()`: Skip whitespace on the current line

logscan.py

```
# - - -   S c a n . d e b l a n k L i n e  
  
def deblankLine ( self ):
```

```
'''Skip blanks on the current line
...'''
```

See Section 8.4, “WHITE\_PATTERN” (p. 9) for the regular expression that matches zero or more whitespace characters.

logscan.py

```
self.tabReMatch ( WHITE_PATTERN )
```

## 36. Scan.match(): Match a specific string

logscan.py

```
# - - - S c a n . m a t c h

def match ( self, s ):
    '''Does the rest of the line start with s?
    ...'''
    if (self.line[self.pos:]).startswith ( s ):
        return self.pos+len(s)
    else:
        return None
```

## 37. Scan.matchArb(): Case-insensitive match

logscan.py

```
# - - - S c a n . m a t c h A r b

def matchArb ( self, s ):
    '''Does the rest of the line start with s, case-insensitive?
    ...'''
```

Conveniently, the Python slice operator does not complain if you specify an ending position beyond the end of the string.

logscan.py

```
#-- 1 --
endPos = self.pos + len(s)

#-- 2 --
# [ t := contents of self.line from position self.pos
#     up to position endPos or end of line, whichever
#     is shorter ]
t = self.line[self.pos:endPos]

#-- 3 --
if s.upper() == t.upper():
    return endPos
else:
    return None
```

## 38. Scan.tabMatch(): Advance if there is a match

logscan.py

```
# - - -   S c a n . t a b M a t c h

def tabMatch ( self, s ):
    '''If s matches at the current position, advance past it.
    ...
    #-- 1 --
    # [ if the current position starts with s ->
    #     endPos := the position just after the match
    #   else ->
    #     endPos := none ]
    endPos = self.match ( s )

    #-- 2 --
    # [ if endPos is None ->
    #     return None
    #   else ->
    #     self := self advanced to endPos
    #     return current line between self.pos and endPos ]
    if endPos is None:
        return None
    else:
        return self.tab ( endPos )
```

## 39. Scan.tabMatchArb(): Case-insensitive match and move

logscan.py

```
# - - -   S c a n . t a b M a t c h A r b

def tabMatchArb ( self, s ):
    '''Check to see if s is next, and advance past it if so.
    ...
    #-- 1 --
    # [ if the current position starts with s, case-insensitive ->
    #     endPos := the position just after the match
    #   else ->
    #     endPos := none ]
    endPos = self.matchArb ( s )

    #-- 2 --
    # [ if endPos is None ->
    #     return None
    #   else ->
    #     self := self advanced to endPos
    #     return current line between self.pos and endPos ]
    if endPos is None:
        return None
    else:
        return self.tab ( endPos )
```

## 40. Scan.reMatch(): Match a regular expression

logscan.py

```
# - - -   S c a n . r e M a t c h

def reMatch ( self, r ):
    '''See if the text at self.pos matches a regular expression.
    ...
    #-- 1 --
    # [ if (r is a string) and (regex (r) matches self.line
    #   starting at self.pos ->
    #     m := a MatchObject representing that match
    #   else if compiled regex (r) matches self.line starting
    #     at self.pos ->
    #     m := a MatchObject representing that match
    #   else ->
    #     m := None ]
```

There is an odd asymmetry in the `re` package: the `.match()` method on a compiled RE has optional arguments to specify starting and ending locations in the source string, but the `re.match()` function does not. My first try at the second line below was “`m = re.match ( r, self.line, self.pos )`”, which fails.

logscan.py

```
if isinstance(r, basestring):
    m = re.match ( r, self.line[self.pos:] )
else:
    m = r.match ( self.line, self.pos )

#-- 2 --
return m
```

## 41. Scan.tabReMatch(): Match and advance the position

logscan.py

```
# - - -   S c a n . t a b R e M a t c h

def tabReMatch ( self, r ):
    '''If the text at self.pos matches, advance
    ...
    #-- 1 --
    # [ if (r is a string) and (regex (r) matches self.line
    #   starting at self.pos ->
    #     m := a MatchObject representing that match
    #   else if compiled regex (r) matches self.line starting
    #     at self.pos ->
    #     m := a MatchObject representing that match
    #   else ->
    #     m := None ]
    m = self.reMatch ( r )

    #-- 2 --
    # [ if m is not None ->
```

```

#         self.pos += length of matched string in m ]
if m is not None:
    self.pos += (m.end() - m.start())

#-- 3 --
return m

```

## 42. Scan.integer(): Parse an integer

logscan.py

```

# - - -   S c a n . i n t e g e r

def integer ( self, maxLen=None ):
    '''Parse a whole number.
    ...

```

For the regular expression used in this logic, see Section 8.5, “INT\_PATTERN” (p. 9).

logscan.py

```

#-- 1 --
# [ if the line in self starts with a string that matches
#   INT_PATTERN ->
#     m := a MatchObject representing the matching text
#   else ->
#     m := None ]
m = self.reMatch ( INT_PATTERN )

#-- 2 --
# [ if m is None ->
#   return None
#   else ->
#     rawInt := matching text from m ]
if m is None:
    return None
else:
    rawInt = m.group()

```

At this point, `m.group()` has the longest matching integer. If the caller specified a `maxLen`, we must truncate it.

logscan.py

```

#-- 3 --
if ( (maxLen is not None) and
      (len(rawInt) > maxLen) ):
    rawInt = rawInt[:maxLen]

#-- 4 --
return int(rawInt)

```

## 43. Scan.fixed(): Parse a float constant

logscan.py

```
# - - -   S c a n . f i x e d

def fixed ( self ):
    '''Parse a fixed-point constant.
    ...
```

For the regular expression used in this logic, see Section 8.6, “FLOAT\_PATTERN” (p. 9).

logscan.py

```
#-- 1 --
# [ if the line in self starts with a string that matches
#   FLOAT_PATTERN ->
#     self := self advanced past the match
#     m := a MatchObject representing the match
#   else ->
#     rawFloat := None ]
m = self.tabReMatch ( FLOAT_PATTERN )

#-- 2 --
if m is None:
    return None
else:
    return float(m.group())
```

## 44. Scan.flatInt(): Parse a fixed-size integer field

logscan.py

```
# - - -   S c a n . f l a t I n t

def flatInt ( self, n ):
    '''Parse a fixed-field integer of size n
    ...
```

This method uses the same regular expression as Section 42, “Scan.integer(): Parse an integer” (p. 31); see Section 8.5, “INT\_PATTERN” (p. 9). First we make sure that there are at least  $n$  characters left on the line. If so, they are matched against INT\_PATTERN; if there is a match, it must match all  $n$  characters.

logscan.py

```
#-- 1 --
# [ if the current line has at least n characters left ->
#   rawInt := the next n characters from the line
#   else ->
#     return None ]
if len(self.line) - self.pos < n:
    return None
else:
    rawInt = self.line[self.pos:self.pos+n]

#-- 2 --
# [ if rawInt matches INT_PATTERN ->
```

```

#     matchLen := the length of the matching part
#     else -> return None ]
m = INT_PATTERN.match ( rawInt )
if m is None:
    return None
else:
    matchLen = m.end() - m.start()

#-- 3 --
if matchLen < n:
    return None
else:
    return int(rawInt)

```

## 45. The singleton class: A classic design pattern

The movement to codify standard design patterns in software architecture stems from this seminal work:

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns. Addison-Wesley, 1995, ISBN 0-201-63361-2.

The purpose of the Singleton design pattern is to make sure that a class has only instance. The Log object described in Section 3, “The singleton LOG object” (p. 3) is an application of this pattern.

This code is in a separate file, `singleton.py`, and not in the `logscan.py` file, because it may be useful elsewhere.

A Python implementation of the Singleton pattern is given on page 84 of this work:

Martelli, Alex. Python in a nutshell. O'Reilly, 2003, 1st ed., ISBN 0-596-00188-6.

The present work is basically the same code with literate annotation.

### 45.1. Code prologue for `singleton.py`

`singleton.py`

```

'''singleton.py: A Python implementation of the Singleton pattern.

Do not edit this file. It is extracted automatically from the
documentation:
    http://www.nmt.edu/tcc/help/lang/python/examples/logscan/
...

```

### 45.2. class Singleton

`singleton.py`

```

# - - - - - c l a s s   S i n g l e t o n

class Singleton(object):
    '''Base class for singleton objects.

```

```

State/Invariants:
Singleton.__classMap:
    [ a dictionary whose keys are the classes
      instantiated so far, and each related value
      is the single instance of that class ]
...

```

This class can manage any number of different derived classes. A class variable, `__classMap`, is used to keep track of their instances.

singleton.py

```
__classMap = {}
```

### 45.3. The `__new__()` method

singleton.py

```

# - - - S i n g l e t o n . _ _ n e w _ _
def __new__ ( cls, *args, **kw ):
    '''Instance factory.

    [ if cls is a key in Singleton.__classMap ->
      return the related value
      else ->
        Singleton.__classMap[cls] := a new
        object of class cls
      return that new object ]
    ...

```

This method is the factory method called whenever a subclass is instantiated. The method of this name is automatically a class method, so its first argument `cls` is the class being instantiated.

- If we have never seen this class before, `cls` will not be a key in the class variable `__classMap`. We will use the `object.__new__` method to create a new instance, then add that class and instance to `__classMap`.
- If we have seen this class before, the class will be a key in the `__classMap` dictionary.

singleton.py

```

#- - 1 - -
try:
    return Singleton.__classMap[cls]
except KeyError:
    inst = Singleton.__classMap[cls] = object.__new__(cls)
    return inst

```

## 46. Test driver for singleton

singtest

```

#!/usr/bin/env python
#=====
# Test driver for the Singleton class.
#
# Do not edit this file. It is extracted automatically from the

```

```

# documentation:
#   http://www.nmt.edu/tcc/help/lang/python/examples/logscan/
#-----

from singleton import *

# - - -   m a i n

def main():
    '''Test the Singleton class.
    ...
    d1 = D('one')
    print d1
    d2 = D('two')
    print d2

# - - - - -   c l a s s   D

class D(Singleton):
    '''Singleton test object.

    Exports:
    D(s):
        [ s is some string ->
          if D has been instantiated before ->
            return the previous instance with its
              .last attribute set to s
          else ->
            return a new instance of D with its .first
              and .last attributes set to s ]
    .first: [ argument to D() on first call ]
    .last:  [ argument to D() on latest call ]
    .__str__(self): [ display self as a string ]
    ...
    everCalled = False

    def __init__( self, s ):
        if D.everCalled:
            self.last = s
        else:
            D.everCalled = True
            self.first = self.last = s

    def __str__( self ):
        return "D.first='%s' .last='%s'" % (self.first, self.last)

#=====
# Epilogue
#-----
if __name__ == '__main__':
    main()

```

