Scanning objects for Python: `Scan`, `Log`, and `Cset`

New Mexico Tech
Computer Center www.nmt.edu/tcc

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.
- The `Cset` class describes sets of characters, and is used by the `Scan` class.

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 a log file.
- 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.

An earlier version of this object, written in the Icon language, satisfied these requirements, but it required that an object be instantiated once and then passed to every routine that might need to issue error messages. Fortunately, the emerging science of software design patterns suggests a way to make it unnecessary to pass the object everywhere it may be needed.

The book *Design Patterns* by Gamma et al describes a pattern called Singleton with these characteristics:

- The first caller to instantiate a Singleton class gets a new object.
- All further callers that instantiate the class get the identical object that was returned to the first caller.

This pattern is ideal for the problem of centralizing error writing and counting.

Methods on the `Log()` *object*

`Log()`

To instantiate the Log object, use a constructor with no arguments. The first caller will get a new instance; all remaining callers will get the same instance returned to the first caller.

`.addLogFile (` $fileName$ `)`

If $fileName$ is a string naming a file that can be opened for writing, that file is created empty and added to the list of files that will receive all messages sent to the Log object.

`.setPrefix (` s `)`

Sets the standard message prefix to s; the default value is `"*** "`. The standard message prefix is automatically prepended to all messages written via the `.error()` and `.msgKind()` methods.

`.error (` s_0 `[ ,` s_1 `] ... )`

Sends an error message to the standard error stream and also to any files added with the `.addLogFile()` method. The message consists of the standard error prefix (see `.setPrefix()`) followed by the concatenation of all the strings passed to the method. Example:

```
Log().error ( "You must have ", `N_ARGS`, " arguments." )
```

`.warning (` s_0 `[ ,` s_1 `] ... )`

Like `.error()`, but sends a warning message. To retrieve the warning count, use `.count(WARNING_KIND)`. Example:

```
Log().warning ( "Your mileage may vary." )
```

`.msgKind (` $kind$`,` s_0 `[ ,` s_1 `] ... )`

If you would like to classify your messages into more types than just "errors," use this method to issue non-error messages. The $kind$ argument is a string that describes the class of messages. For example:

```
Log().msgKind ( "Warning",
                "Objects in mirror are further than they appear." )
```

`.count ( kind=None )`

This method returns the number of errors of a given kind that have been sent to the Log object. If the argument is omitted, the number of errors is returned. To find the error count for another kind of message, pass in the same argument you passed to the first argument of `.msgKind()`. For example:

```
Log().message ( "There were %d errors and %d warnings." %
                ( Log().count(), Log().count ( "Warning" ) ) )
```

`.fatal ( s0 [ , s1 ] ... )`

If you want to write a message and then terminate execution, use this method. The message is made up of the arguments, concatenated.

This method does not return to the caller. After writing the message, it raises `SystemExit`.

`.message ( s0 [ , s1 ] ... )`

Sends the concatenation of its arguments to the standard error stream and any other files added through `.addLogFile()`, but does not increment any counts of errors by class. Useful for multiline error messages. Prepends the standard error prefix.

`.write ( s0 [ , s1 ] ... )`

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

The `Scan` *object*

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.

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.

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.

The `Scan` object also supports an optional line-comment character. For example, you might ask for the `Scan` object 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.

Basic methods

To scan a file named f using the Scan object, instantiate your Scan object like this:

```
from scan import *
scan = Scan ( fileName, commentChar=None, callback=None )
```

If the file is readable, the `Scan()` constructor will return a Scan object positioned at the start of the file, otherwise it will raise `IOError`. Arguments:

fileName	The name of the file to be scanned.
`commentChar`	If given, this argument should be a string containing the comment character. The default is not to have a comment character.
`callback`	If given, this is a procedure with calling sequence `callback(scan)`, where `scan` is the `Scan` object, that returns a string that is appended to an error message each time one is issued.

The usual method of looping over the lines of the file is to use something like this:

```
while not scan.atEndFile():
    (code to process one line)
    scan.nextLine()
```

The `.atEndFile()` method returns true when the input is exhausted; the `.nextLine()` method attempts to advance to the next line if there is one.

Within each line, you can use a wide assortment of scanning primitives to move and scan over items from the stream.

The `Scan` object makes heavy use of the `Log` object to manage the error messages that you may issue. This object is described above.

`.atEndFile`

This is a member that always equals:

- 1 if scan is exhausted,
- 0 if any content remains.

`.close()`

This method should be called when scanning is completed:

```
scan.close ( )
```

`.atEndLine()`

Returns:

- 1 if the current line is exhausted,
- 0 if any characters remain on the current line.

`.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()` predicated described above.

`.pos`

This member gives the position on the current line, starting from 0. It is a **read-only** member.

`.line`

This member contains the text of the current line, with any trailing comment trimmed if there is one. It is a **read-only** member.

Error signaling and message methods

`.error (` s_0 `[,` s_1`] ...)`

To issue an ordinary error message, use this method and pass it one or more strings. If this is the first error on the current line, the line will be echoed to the `Log()` singleton. Then a line will be written with a caret (^) under the echoed line at the current scan position. This will be followed by a line starting with `*** Error:` followed by the strings you passed in, concatenated. The `Log()` object will count this as an error.

To issue a multi-line error message, use `.error()` for the first line and `.message()`, described below, for succeeding lines.

Here's an example:

```
scan.error ( "I have now taunted you ", `nTimes`, " times." );
```

`.warning (` s_0 `[,` s_1`] ...)`

Same as `.error()`, but sends a warning message. To retrieve a count of such messages, use `Log().count(WARNING_KIND)`.

`.msgKind (` *kind*`,` s_0 `[ ,` s_1 `] ... )`

Normally all errors are prefaced by the string `"Error:"`. However, you can define multiple kinds of errors—for example, errors, warnings, and informational messages—and keep separate counts of the errors in each category. To issue a message in a non-error category, use this method, where:

`kind`	is a string that describes the error category.
s_i	The remaining arguments are concatenated to form the error message.

To issue a warning, for example, you might use a call like this:

```
scan.msgKind ( "Warning:  ", "Your mileage may vary." )
```

`.message (` s_0 `[,` s_1 `] ... )`

The arguments are concatenated into a single line and sent to the `Log()` object. The standard message prefix, `Scan.MESSAGE_PREFIX`, is prepended to the line.

`.write (` s_0 `[,` s_1 `] ... )`

Works like the `.message()` method, but does not prepend the standard message prefix.

`.msgCount ( [ kind ] )`

If the $kind$ argument is omitted, the error count from `Log()` is returned as an integer. If there have been no errors since this object was instantiated, you will get a zero back.

To retrieve the counts of other kinds of errors, pass in the same string you used with the `.msgKind()` method.

Note that the scan object does not keep its own error counts, so any errors sent to the `Log()` object through other agencies will affect this count.

Application. The obvious application for `.msgCount()` is printing messages such as "You had 9 errors and 3 warnings." However, sometimes I find it convenient for diffusing error reporting while centralizing error detection.

For example, suppose you have a large section of code that checks for a bunch of different errors, and any one of several dozen procedures might generate an error. One way to handle it would be for each procedure to fail if it detected an error. There are two disadvantages to this:

- Every call to a procedure must check for failure and take appropriate action.
- If you fail on the first error, you may miss a chance to detect other errors.

So another way to handle this is to record the error count before executing that section of code, and then check the error count afterwards to see if it has gone up. If so, you know there has been an error somewhere. Here is how that code might look:

```
def Sample(scan):
  errCount = scan.msgCount()
  (some complex operation that might generate errors anywhere)
  if errCount < scan.msgCount():
    (handle the presence of errors)
```

Basic scanning primitives

The methods below are provided to handle common operations.

`.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 )`

This method is used to try to jump the position within the current line to a new position p. If it succeeds, 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. Position -1 is at the end of the string; -2 is 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 )`

This method is 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, you could use the expression `scan.isPos(-2)`, which would return 1 if you are positioned on the last character, 0 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`.

`.upto ( c )`

This method is used to look for a character of a particular kind on the current line at or after the current position. The argument must be a `Cset` object (see below).

If there are any characters remaining on the current line that are in the `Cset` c, this method returns the index on the current line of the first such character. If no characters remaining on the current line are in c, the method returns `None`.

`.deblank()`

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()`

Like `.deblank()`, but never advances further than the end of the current line.

`.any ( c )`

Checks to see if the next character is in a given class c. The argument must be a `Cset`.If there is at least one character on the current line and that character is in c, returns the index on the current line of the character *following* the next character; otherwise, it returns `None`.

`.tabAny ( c )`

If there is at least one character left on the current line, and that character is in the `Cset` c, the position is advanced by one and the matching character is returned; otherwise returns `None`.

`.many ( c )`

If there are one or more characters on the current line starting at the current position, returns the index on the current line of the first character *beyond* those characters; otherwise returns `None`.

`.tabMany ( c )`

If there are one or more characters on the current line starting at the current position, all of which characters are in the `Cset` c, the position is advanced past those characters and the string of all matching characters is returned; otherwise returns `None`.

`.match ( s )`

The argument must be a nonempty string. If the remainder of the current line starts with s, returns the position of the character just past the matching part; otherwise returns `None`.

`.matchArb ( s )`

Like `.match()`, but ignores case. That is, if s with all its letters upshifted matches a string at the beginning of the remainder of the current line with all of its letters upshifted, returns the position of the character following the match; otherwise returns `None`.

`.tabMatch ( s )`

If there is a string at the beginning of the remainder of the current line that matches s, the position is advanced by the length of s and the matching string is returned. If there is no match, returns `None`.

`.tabMatchArb ( s )`

Like `.tabMatch()`, but uses case-insensitive comparison like `.matchArb()`.

`.reMatch ( r )`

Works like `.match()`, but matches a regular expression. The argument r can be a raw string like the first argument of the `.match()` method in the Python `re` module. You may also compile the pattern using `re.compile()` and pass the result of that compilation to the r argument. If there is a match, this method returns the usual `MatchObject` instance as in `re.match()`; otherwise it returns `None`.

`.tabReMatch ( r )`

The argument is either a raw string or a compiled regular expression, as in the `.reMatch()` method above. If the remainder of the current line starts with a string matching r, the position is advanced past the matching string, and a `MatchObject` instance is returned, as in the standard Python `re` module.

Advanced scanning primitives

The methods below deal with somewhat larger scannable objects.

`.integer ( maxLen=None )`

To parse an integer (that is, a string of digits), use this method. Use the default `maxLen` if you don't care how long the integer is. If the integer is not to exceed a given number of digits, pass in the maximum field length.

If the remainder of the current line starts with at least one digit but not more than `maxLen` (defaulting to infinity), the position is advanced past leading digits and those digits are returned *as an integer*, not a string. If the string of digits is there but too long, an error will be sent to the `Log` object, the position will be advanced by `maxLen` digits, and those digits will be returned as integer type. If the next character is not a digit, returns `None`.

`.fixed ( )`

Scans for a fixed-point number. If the input starts with at least one digit, optionally preceded by a minus sign, and optionally followed by a decimal point and optional trailing digits, the position is advanced past all that, and the matching string is returned *as a string*. If the input does not start with at least one digit, returns `None`.

Note that this does not allow fixed-point numbers to start with a decimal point, so the string `".1"` would not be considered valid.

`.flatCset (` n`,` c `)`

Checks to see if the input begins with a fixed-size field of size n consisting entirely of characters from `Cset` c. If so, advances the position by n and returns the contents of the field as a string. If there are fewer than n characters remaining on the current line, or some of them are not in c, returns `None`.

`.flatInt (` n `)`

Checks to see if the input begins with an integer field of size n. A valid field must have the number right-aligned in that field, although the optional minus sign can occur anywhere in the leading blanks. If the input matches, the position is advanced by n and the match returned as type integer; otherwise returns `None`.

The `Cset` *class*

This class is modeled after the built-in `cset` type in Icon. The purpose of a `Cset` object is to describe some subset of the ASCII character set. It is used by a number of scanning methods in the `Scan` object described above.

For example, suppose that you want to define a variable name as a string consisting of letters, digits, and the underscore (_) character. You can define a `Cset` as consisting of just those characters.

`Cset (` s `)`

To instantiate a Cset, call the constructor with a string containing all the characters that you want to be in the set. For example:

```
import string
varCset = Cset ( string.letters + string.digits + "_" )
```

This example would instantiate a `Cset` consisting of all the letters and digits plus the underscore character.

To continue the example, suppose you are scanning a line with a `Scan` object named `scan`, and you expect a variable name made up of these characters as the next item on the line. Here is a code fragment that stores the variable name in `v` if there is a match, or writes an error message if not:

```
v = scan.tabMany ( varCset )
if not v:
  scan.error ( "Expecting a variable name" )
```

`.has ( c )`

This method tests a one-character string c to see if it is in the set. It returns 1 if c is in self, 0 otherwise.

`.__str__()`

This method allows you to use the built-in `str(c)` function on a `Cset` c. It returns a string of all the characters in c, in ascending order by their ASCII value.

`.union ( d )`

Returns a new `Cset` consisting of all the characters that are either in self or in another `Cset` d.

`.intersect ( d )`

Returns a new `Cset` consisting of all the characters that are in self and also in another `Cset` d.

`.delta ( d )`

Returns a new `Cset` containing all the characters that are in self but not in another `Cset` d.

`.complement()`

Returns a new `Cset` consisting of all the ASCII characters that are not in self. For the purposes of this method, ASCII is assumed to be an 8-bit code.

`.ucase`

`.lcase`

`.letters`

`.digits`

`.whitespace`

`.universe`

These members represent predefined `Cset` objects for some common cases. The `.ucase` and `.lcase` members are the uppercase and lowercase letters, respectively, and `.letters` is all the letters. The `.digits` member is the `Cset` of digits. `.whitespace` is the set of whitespace characters (space, newline, return, horizontal tab, and formfeed). Finally, `.universe` is the `Cset` of all characters.