Interfaces ~
|if_cscop.txt| using Cscope with Vim
|if_lua.txt| Lua interface
|if_mzsch.txt| MzScheme interface
|if_perl.txt| Perl interface
|if_pyth.txt| Python interface
|if_tcl.txt| Tcl interface
|if_ole.txt| OLE automation interface for Win32
|if_ruby.txt| Ruby interface
|debugger.txt| Interface with a debugger
|workshop.txt| Sun Visual Workshop interface
|netbeans.txt| NetBeans External Editor interface
|sign.txt| debugging signs

==
if_cscop.txt For Vim version 8.0. Last change: 2017 Jun 14

                  VIM REFERENCE MANUAL   by Andy Kahn

                                                        cscope *Cscope*
This document explains how to use Vim's cscope interface.

Cscope is a tool like ctags, but think of it as ctags on steroids since it
does a lot more than what ctags provides. In Vim, jumping to a result from
a cscope query is just like jumping to any tag; it is saved on the tag stack
so that with the right keyboard mappings, you can jump back and forth between
functions as you normally would with |tags|.

This is currently for Unix and Win32 only.
{Vi does not have any of these commands}

==
1. Cscope introduction   *cscope-intro*

The following text is taken from a version of the cscope man page:

  Cscope is an interactive screen-oriented tool that helps you:

       Learn how a C program works without endless flipping through a thick
       listing.

       Locate the section of code to change to fix a bug without having to
       learn the entire program.

       Examine the effect of a proposed change such as adding a value to an
       enum variable.

       Verify that a change has been made in all source files such as adding
       an argument to an existing function.

       Rename a global variable in all source files.

```
       Change a constant to a preprocessor symbol in selected lines of files.

  It is designed to answer questions like:
       Where is this symbol used?
       Where is it defined?
       Where did this variable get its value?
       What is this global symbol's definition?
       Where is this function in the source files?
       What functions call this function?
       What functions are called by this function?
       Where does the message "out of space" come from?
       Where is this source file in the directory structure?
       What files include this header file?

  Cscope answers these questions from a symbol database that it builds the
  first time it is used on the source files.  On a subsequent call, cscope
  rebuilds the database only if a source file has changed or the list of
  source files is different.  When the database is rebuilt the data for the
  unchanged files is copied from the old database, which makes rebuilding
  much faster than the initial build.

                                     -----

When cscope is normally invoked, you will get a full-screen selection
screen allowing you to make a query for one of the above questions.
However, once a match is found to your query and you have entered your
text editor to edit the source file containing match, you cannot simply
jump from tag to tag as you normally would with vi's Ctrl-] or :tag
command.

Vim's cscope interface is done by invoking cscope with its line-oriented
interface, and then parsing the output returned from a query.  The end
result is that cscope query results become just like regular tags, so
you can jump to them just like you do with normal tags (Ctrl-] or :tag)
and then go back by popping off the tagstack with Ctrl-T.  (Please note
however, that you don't actually jump to a cscope tag simply by doing
Ctrl-] or :tag without remapping these commands or setting an option.
See the remaining sections on how the cscope interface works and for
suggested use.)


==============================================================================
2. Cscope related commands                                  *cscope-commands*

                *:cscope* *:cs* *:scs* *:scscope* *E259* *E262* *E561* *E560*
All cscope commands are accessed through suboptions to the cscope commands.
        `:cscope` or `:cs` is the main command
        `:scscope` or `:scs` does the same and splits the window
        `:lcscope` or `:lcs` uses the location list, see |:lcscope|

The available subcommands are:

                        *E563* *E564* *E566* *E568* *E622* *E623* *E625*
                        *E626* *E609*
    add   : Add a new cscope database/connection.

        USAGE   :cs add {file|dir} [pre-path] [flags]

            [pre-path] is the pathname used with the -P command to cscope.

            [flags] are any additional flags you want to pass to cscope.
```

```
        EXAMPLES >
            :cscope add /usr/local/cdb/cscope.out
            :cscope add /projects/vim/cscope.out /usr/local/vim
            :cscope add cscope.out /usr/local/vim -C
<
                                        *cscope-find* *cs-find* *E567*
    find  : Query cscope.  All cscope query options are available
            except option #5 ("Change this grep pattern").

        USAGE   :cs find {querytype} {name}

            {querytype} corresponds to the actual cscope line
            interface numbers as well as default nvi commands:

                0 or s: Find this C symbol
                1 or g: Find this definition
                2 or d: Find functions called by this function
                3 or c: Find functions calling this function
                4 or t: Find this text string
                6 or e: Find this egrep pattern
                7 or f: Find this file
                8 or i: Find files #including this file
                9 or a: Find places where this symbol is assigned a value

        For all types, except 4 and 6, leading white space for {name} is
        removed.  For 4 and 6 there is exactly one space between {querytype}
        and {name}.  Further white space is included in {name}.

        EXAMPLES >
            :cscope find c vim_free
            :cscope find 3  vim_free
<
            These two examples perform the same query: functions calling
            "vim_free". >

            :cscope find t initOnce
            :cscope find t  initOnce
<
            The first one searches for the text "initOnce", the second one for
            " initOnce". >

            :cscope find 0 DEFAULT_TERM
<
            Executing this example on the source code for Vim 5.1 produces the
            following output:

            Cscope tag: DEFAULT_TERM
               #   line  filename / context / line
               1   1009  vim-5.1-gtk/src/term.c <<GLOBAL>>
                         #define DEFAULT_TERM (char_u *)"amiga"
               2   1013  vim-5.1-gtk/src/term.c <<GLOBAL>>
                         #define DEFAULT_TERM (char_u *)"win32"
               3   1017  vim-5.1-gtk/src/term.c <<GLOBAL>>
                         #define DEFAULT_TERM (char_u *)"pcterm"
               4   1021  vim-5.1-gtk/src/term.c <<GLOBAL>>
                         #define DEFAULT_TERM (char_u *)"ansi"
               5   1025  vim-5.1-gtk/src/term.c <<GLOBAL>>
                         #define DEFAULT_TERM (char_u *)"vt52"
               6   1029  vim-5.1-gtk/src/term.c <<GLOBAL>>
                         #define DEFAULT_TERM (char_u *)"os2ansi"
               7   1033  vim-5.1-gtk/src/term.c <<GLOBAL>>
```

```
                      #define DEFAULT_TERM (char_u *)"ansi"
               8   1037  vim-5.1-gtk/src/term.c <<GLOBAL>>
                      # undef DEFAULT_TERM
               9   1038  vim-5.1-gtk/src/term.c <<GLOBAL>>
                      #define DEFAULT_TERM (char_u *)"beos-ansi"
              10   1042  vim-5.1-gtk/src/term.c <<GLOBAL>>
                      #define DEFAULT_TERM (char_u *)"mac-ansi"
              11   1335  vim-5.1-gtk/src/term.c <<set_termname>>
                      term = DEFAULT_TERM;
              12   1459  vim-5.1-gtk/src/term.c <<set_termname>>
                      if (STRCMP(term, DEFAULT_TERM))
              13   1826  vim-5.1-gtk/src/term.c <<termcapinit>>
                      term = DEFAULT_TERM;
              14   1833  vim-5.1-gtk/src/term.c <<termcapinit>>
                      term = DEFAULT_TERM;
              15   3635  vim-5.1-gtk/src/term.c <<update_tcap>>
                      p = find_builtin_term(DEFAULT_TERM);
          Enter nr of choice (<CR> to abort):

          The output shows several pieces of information:
          1. The tag number (there are 15 in this example).
          2. The line number where the tag occurs.
          3. The filename where the tag occurs.
          4. The context of the tag (e.g., global, or the function name).
          5. The line from the file itself.

  help  : Show a brief synopsis.

          USAGE   :cs help

                                                          *E261*
  kill  : Kill a cscope connection (or kill all cscope connections).

          USAGE   :cs kill {num|partial_name}

          To kill a cscope connection, the connection number or a partial
          name must be specified.  The partial name is simply any part of
          the pathname of the cscope database.  Kill a cscope connection
          using the partial name with caution!

          If the specified connection number is -1, then _ALL_ cscope
          connections will be killed.

  reset : Reinit all cscope connections.

          USAGE   :cs reset

  show  : Show cscope connections.

          USAGE   :cs show

                                                          *:lcscope* *:lcs*
This command is same as the ":cscope" command, except when the
'cscopequickfix' option is set, the location list for the current window is
used instead of the quickfix list to show the cscope results.

                                                          *:cstag* *E257* *E562*
If you use cscope as well as ctags, |:cstag| allows you to search one or
the other before making a jump.  For example, you can choose to first
search your cscope database(s) for a match, and if one is not found, then
your tags file(s) will be searched.  The order in which this happens
is determined by the value of |csto|.  See |cscope-options| for more
```

details.

|:cstag| performs the equivalent of ":cs find g" on the identifier when
searching through the cscope database(s).

|:cstag| performs the equivalent of |:tjump| on the identifier when searching
through your tags file(s).

==
3. Cscope options *cscope-options*

Use the |:set| command to set all cscope options. Ideally, you would do
this in one of your startup files (e.g., .vimrc). Some cscope related
variables are only valid within |.vimrc|. Setting them after vim has
started will have no effect!

cscopeprg *csprg*
'cscopeprg' specifies the command to execute cscope. The default is
"cscope". For example: >
:set csprg=/usr/local/bin/cscope
<
cscopequickfix *csqf* *E469*
{not available when compiled without the |+quickfix| feature}
'cscopequickfix' specifies whether to use quickfix window to show cscope
results. This is a list of comma-separated values. Each item consists of
|cscope-find| command (s, g, d, c, t, e, f, i or a) and flag (+, - or 0).
'+' indicates that results must be appended to quickfix window,
'-' implies previous results clearance, '0' or command absence - don't use
quickfix. Search is performed from start until first command occurrence.
The default value is "" (don't use quickfix anyway). The following value
seems to be useful: >
:set cscopequickfix=s-,c-,d-,i-,t-,e-,a-
<
cscopetag *cst*
If 'cscopetag' is set, the commands ":tag" and CTRL-] as well as "vim -t"
will always use |:cstag| instead of the default :tag behavior. Effectively,
by setting 'cst', you will always search your cscope databases as well as
your tag files. The default is off. Examples: >
:set cst
:set nocst
<
cscoperelative *csre*
If 'cscoperelative' is set, then in absence of a prefix given to cscope
(prefix is the argument of -P option of cscope), basename of cscope.out
location (usually the project root directory) will be used as the prefix
to construct an absolute path. The default is off. Note: This option is
only effective when cscope (cscopeprg) is initialized without a prefix
path (-P). Examples: >
:set csre
:set nocsre
<
cscopetagorder *csto*
The value of 'csto' determines the order in which |:cstag| performs a search.
If 'csto' is set to zero, cscope database(s) are searched first, followed
by tag file(s) if cscope did not return any matches. If 'csto' is set to
one, tag file(s) are searched before cscope database(s). The default is zero.
Examples: >
:set csto=0
:set csto=1
<
cscopeverbose *csverb*

```
If 'cscopeverbose' is not set (the default), messages will not be printed
indicating success or failure when adding a cscope database.  Ideally, you
should reset this option in your |.vimrc| before adding any cscope databases,
and after adding them, set it.  From then on, when you add more databases
within Vim, you will get a (hopefully) useful message should the database fail
to be added.  Examples: >
        :set csverb
        :set nocsverb
<
                                                        *cscopepathcomp* *cspc*
The value of 'cspc' determines how many components of a file's path to
display.  With the default value of zero the entire path will be displayed.
The value one will display only the filename with no path.  Other values
display that many components.  For example: >
        :set cspc=3
will display the last 3 components of the file's path, including the file
name itself.

==============================================================================
4. How to use cscope in Vim                             *cscope-howtouse*

The first thing you need to do is to build a cscope database for your
source files.  For the most basic case, simply do "cscope -b".  Please
refer to the cscope man page for more details.

Assuming you have a cscope database, you need to "add" the database to Vim.
This establishes a cscope "connection" and makes it available for Vim to use.
You can do this in your .vimrc file, or you can do it manually after starting
vim.  For example, to add the cscope database "cscope.out", you would do:

        :cs add cscope.out

You can double-check the result of this by executing ":cs show".  This will
produce output which looks like this:

 # pid    database name                       prepend path
 0 28806  cscope.out                          <none>

Note:
Because of the Microsoft RTL limitations, Win32 version shows 0 instead
of the real pid.

Once a cscope connection is established, you can make queries to cscope and
the results will be printed to you.  Queries are made using the command
":cs find".  For example:

        :cs find g ALIGN_SIZE

This can get a little cumbersome since one ends up doing a significant
amount of typing.  Fortunately, there are ways around this by mapping
shortcut keys.  See |cscope-suggestions| for suggested usage.

If the results return only one match, you will automatically be taken to it.
If there is more than one match, you will be given a selection screen to pick
the match you want to go to.  After you have jumped to the new location,
simply hit Ctrl-T to get back to the previous one.


==============================================================================
5. Limitations                                          *cscope-limitations*

Cscope support for Vim is only available on systems that support these four
```

```
system calls: fork(), pipe(), execl(), waitpid().  This means it is mostly
limited to Unix systems.

Additionally Cscope support works for Win32.  For more information and a
cscope version for Win32 see:

        http://iamphet.nm.ru/cscope/index.html

The DJGPP-built version from http://cscope.sourceforge.net is known to not
work with Vim.

Hard-coded limitation: doing a |:tjump| when |:cstag| searches the tag files
is not configurable (e.g., you can't do a tselect instead).

==============================================================================
6. Suggested usage                                      *cscope-suggestions*

Put these entries in your .vimrc (adjust the pathname accordingly to your
setup): >

        if has("cscope")
                set csprg=/usr/local/bin/cscope
                set csto=0
                set cst
                set nocsverb
                " add any database in current directory
                if filereadable("cscope.out")
                    cs add cscope.out
                " else add database pointed to by environment
                elseif $CSCOPE_DB != ""
                    cs add $CSCOPE_DB
                endif
                set csverb
        endif

By setting 'cscopetag', we have effectively replaced all instances of the :tag
command with :cstag.  This includes :tag, Ctrl-], and "vim -t".  In doing
this, the regular tag command not only searches your ctags generated tag
files, but your cscope databases as well.

Some users may want to keep the regular tag behavior and have a different
shortcut to access :cstag.  For example, one could map Ctrl-_  (underscore)
to :cstag with the following command: >

        map <C-_> :cstag <C-R>=expand("<cword>")<CR><CR>

A couple of very commonly used cscope queries (using ":cs find") is to
find all functions calling a certain function and to find all occurrences
of a particular C symbol.  To do this, you can use these mappings as an
example: >

        map g<C-]> :cs find 3 <C-R>=expand("<cword>")<CR><CR>
        map g<C-\> :cs find 0 <C-R>=expand("<cword>")<CR><CR>

These mappings for Ctrl-] (right bracket) and Ctrl-\ (backslash) allow you to
place your cursor over the function name or C symbol and quickly query cscope
for any matches.

Or you may use the following scheme, inspired by Vim/Cscope tutorial from
Cscope Home Page (http://cscope.sourceforge.net/): >

        nmap <C-_>s :cs find s <C-R>=expand("<cword>")<CR><CR>
```

```
        nmap <C-_>g :cs find g <C-R>=expand("<cword>")<CR><CR>
        nmap <C-_>c :cs find c <C-R>=expand("<cword>")<CR><CR>
        nmap <C-_>t :cs find t <C-R>=expand("<cword>")<CR><CR>
        nmap <C-_>e :cs find e <C-R>=expand("<cword>")<CR><CR>
        nmap <C-_>f :cs find f <C-R>=expand("<cfile>")<CR><CR>
        nmap <C-_>i :cs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
        nmap <C-_>d :cs find d <C-R>=expand("<cword>")<CR><CR>
        nmap <C-_>a :cs find a <C-R>=expand("<cword>")<CR><CR>

        " Using 'CTRL-spacebar' then a search type makes the vim window
        " split horizontally, with search result displayed in
        " the new window.

        nmap <C-Space>s :scs find s <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space>g :scs find g <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space>c :scs find c <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space>t :scs find t <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space>e :scs find e <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space>f :scs find f <C-R>=expand("<cfile>")<CR><CR>
        nmap <C-Space>i :scs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
        nmap <C-Space>d :scs find d <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space>a :scs find a <C-R>=expand("<cword>")<CR><CR>

        " Hitting CTRL-space *twice* before the search type does a vertical
        " split instead of a horizontal one

        nmap <C-Space><C-Space>s
                \:vert scs find s <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space><C-Space>g
                \:vert scs find g <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space><C-Space>c
                \:vert scs find c <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space><C-Space>t
                \:vert scs find t <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space><C-Space>e
                \:vert scs find e <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space><C-Space>i
                \:vert scs find i ^<C-R>=expand("<cfile>")<CR>$<CR>
        nmap <C-Space><C-Space>d
                \:vert scs find d <C-R>=expand("<cword>")<CR><CR>
        nmap <C-Space><C-Space>a
                \:vert scs find a <C-R>=expand("<cword>")<CR><CR>

==============================================================================
7. Cscope availability and information                  *cscope-info*

If you do not already have cscope (it did not come with your compiler
license or OS distribution), then you can download it for free from:
        http://cscope.sourceforge.net/
This is released by SCO under the BSD license.

If you want a newer version of cscope, you will probably have to buy it.
According to the (old) nvi documentation:

        You can buy version 13.3 source with an unrestricted license
        for $400 from AT&T Software Solutions by calling +1-800-462-8146.

Also you can download cscope 13.x and mlcscope 14.x (multi-lingual cscope
which supports C, C++, Java, lex, yacc, breakpoint listing, Ingres, and SDL)
from World-Wide Exptools Open Source packages page:
        http://www.bell-labs.com/project/wwexptools/packages.html
```

```
In Solaris 2.x, if you have the C compiler license, you will also have
cscope.  Both are usually located under /opt/SUNWspro/bin

SGI developers can also get it.  Search for Cscope on this page:
        http://freeware.sgi.com/index-by-alpha.html
        https://toolbox.sgi.com/toolbox/utilities/cscope/
The second one is for those who have a password for the SGI toolbox.

There is source to an older version of a cscope clone (called "cs") available
on the net.  Due to various reasons, this is not supported with Vim.

The cscope interface/support for Vim was originally written by
Andy Kahn <ackahn@netapp.com>.  The original structure (as well as a tiny
bit of code) was adapted from the cscope interface in nvi.  Please report
any problems, suggestions, patches, et al., you have for the usage of
cscope within Vim to him.
                                                        *cscope-win32*
For a cscope version for Win32 see:
        http://code.google.com/p/cscope-win32/

Win32 support was added by Sergey Khorev <sergey.khorev@gmail.com>.  Contact
him if you have Win32-specific issues.

 vim:tw=78:ts=8:ft=help:norl:
*if_lua.txt*    For Vim version 8.0.  Last change: 2015 Oct 16


                  VIM REFERENCE MANUAL    by Luis Carvalho


The Lua Interface to Vim                                *lua* *Lua*

1. Commands                     |lua-commands|
2. The vim module               |lua-vim|
3. List userdata                |lua-list|
4. Dict userdata                |lua-dict|
5. Funcref userdata             |lua-funcref|
6. Buffer userdata              |lua-buffer|
7. Window userdata              |lua-window|
8. The luaeval function         |lua-luaeval|
9. Dynamic loading              |lua-dynamic|

{Vi does not have any of these commands}

The Lua interface is available only when Vim was compiled with the
|+lua| feature.

==============================================================================
1. Commands                                             *lua-commands*

                                                        *:lua*
:[range]lua {chunk}
                        Execute Lua chunk {chunk}.    {not in Vi}

Examples:
>
        :lua print("Hello, Vim!")
        :lua local curbuf = vim.buffer() curbuf[7] = "line #7"
<

:[range]lua << {endmarker}
{script}
```

```
{endmarker}
                        Execute Lua script {script}.  {not in Vi}
                        Note: This command doesn't work when the Lua
                        feature wasn't compiled in.  To avoid errors, see
                        |script-here|.

{endmarker} must NOT be preceded by any white space.  If {endmarker} is
omitted from after the "<<", a dot '.' must be used after {script}, like
for the |:append| and |:insert| commands.
This form of the |:lua| command is mainly useful for including Lua code
in Vim scripts.

Example:
>
        function! CurrentLineInfo()
        lua << EOF
        local linenr = vim.window().line
        local curline = vim.buffer()[linenr]
        print(string.format("Current line [%d] has %d chars",
                linenr, #curline))
        EOF
        endfunction
<
To see what version of Lua you have: >
        :lua print(_VERSION)

If you use LuaJIT you can also use this: >
        :lua print(jit.version)
<

                                                        *:luado*
:[range]luado {body}    Execute Lua function "function (line, linenr) {body}
                        end" for each line in the [range], with the function
                        argument being set to the text of each line in turn,
                        without a trailing <EOL>, and the current line number.
                        If the value returned by the function is a string it
                        becomes the text of the line in the current turn. The
                        default for [range] is the whole file: "1,$".
                                                        {not in Vi}

Examples:
>
        :luado return string.format("%s\t%d", line:reverse(), #line)

        :lua require"lpeg"
        :lua -- balanced parenthesis grammar:
        :lua bp = lpeg.P{ "(" * ((1 - lpeg.S"()") + lpeg.V(1))^0 * ")" }
        :luado if bp:match(line) then return "-->\t" .. line end
<

                                                        *:luafile*
:[range]luafile {file}
                        Execute Lua script in {file}. {not in Vi}
                        The whole argument is used as a single file name.

Examples:
>
        :luafile script.lua
        :luafile %
<

All these commands execute a Lua chunk from either the command line (:lua and
```

:luado) or a file (:luafile) with the given line [range]. Similarly to the Lua interpreter, each chunk has its own scope and so only global variables are shared between command calls. All Lua default libraries are available. In addition, Lua "print" function has its output redirected to the Vim message area, with arguments separated by a white space instead of a tab.

Lua uses the "vim" module (see |lua-vim|) to issue commands to Vim and manage buffers (|lua-buffer|) and windows (|lua-window|). However, procedures that alter buffer content, open new buffers, and change cursor position are restricted when the command is executed in the |sandbox|.

==
2. The vim module *lua-vim*

Lua interfaces Vim through the "vim" module. The first and last line of the input range are stored in "vim.firstline" and "vim.lastline" respectively. The module also includes routines for buffer, window, and current line queries, Vim evaluation and command execution, and others.

vim.list([arg]) Returns an empty list or, if "arg" is a Lua table with numeric keys 1, ..., n (a "sequence"), returns a list l such that l[i] = arg[i] for i = 1, ..., n (see |List|). Non-numeric keys are not used to initialize the list. See also |lua-eval| for conversion rules. Example: >
:lua t = {math.pi, false, say = 'hi'}
:echo luaeval('vim.list(t)')
:" [3.141593, 0], 'say' is ignored
<

vim.dict([arg]) Returns an empty dictionary or, if "arg" is a Lua table, returns a dict d such that d[k] = arg[k] for all string keys k in "arg" (see |Dictionary|). Number keys are converted to strings. Keys that are not strings are not used to initialize the dictionary. See also |lua-eval| for conversion rules. Example: >
:lua t = {math.pi, false, say = 'hi'}
:echo luaeval('vim.dict(t)')
:" {'say': 'hi'}, numeric keys ignored
<

vim.funcref({name}) Returns a Funcref to function {name} (see |Funcref|). It is equivalent to Vim's "function". NOT IMPLEMENTED YET

vim.buffer([arg]) If "arg" is a number, returns buffer with number "arg" in the buffer list or, if "arg" is a string, returns buffer whose full or short name is "arg". In both cases, returns 'nil' (nil value, not string) if the buffer is not found. Otherwise, if "toboolean(arg)" is 'true' returns the first buffer in the buffer list or else the current buffer.

vim.window([arg]) If "arg" is a number, returns window with number "arg" or 'nil' (nil value, not string) if not found. Otherwise, if "toboolean(arg)" is 'true' returns the first window or else the current window.

vim.type({arg}) Returns the type of {arg}. It is equivalent to

```
                                Lua's "type" function, but returns "list",
                                "dict", "funcref", "buffer", or "window" if
                                {arg} is a list, dictionary, funcref, buffer,
                                or window, respectively. Examples: >
                                        :lua l = vim.list()
                                        :lua print(type(l), vim.type(l))
                                        :" userdata list
<
        vim.command({cmd})      Executes the vim (ex-mode) command {cmd}.
                                Examples: >
                                        :lua vim.command"set tw=60"
                                        :lua vim.command"normal ddp"
<
        vim.eval({expr})        Evaluates expression {expr} (see |expression|),
                                converts the result to Lua, and returns it.
                                Vim strings and numbers are directly converted
                                to Lua strings and numbers respectively. Vim
                                lists and dictionaries are converted to Lua
                                userdata (see |lua-list| and |lua-dict|).
                                Examples: >
                                        :lua tw = vim.eval"&tw"
                                        :lua print(vim.eval"{'a': 'one'}".a)
<
        vim.line()              Returns the current line (without the trailing
                                <EOL>), a Lua string.

        vim.beep()              Beeps.

        vim.open({fname})       Opens a new buffer for file {fname} and
                                returns it. Note that the buffer is not set as
                                current.


==============================================================================
3. List userdata                                        *lua-list*

List userdata represent vim lists, and the interface tries to follow closely
Vim's syntax for lists. Since lists are objects, changes in list references in
Lua are reflected in Vim and vice-versa. A list "l" has the following
properties and methods:

Properties
----------
        o "#l" is the number of items in list "l", equivalent to "len(l)"
            in Vim.
        o "l[k]" returns the k-th item in "l"; "l" is zero-indexed, as in Vim.
            To modify the k-th item, simply do "l[k] = newitem"; in
            particular, "l[k] = nil" removes the k-th item from "l".
        o "l()" returns an iterator for "l".

Methods
-------
        o "l:add(item)" appends "item" to the end of "l".
        o "l:insert(item[, pos])" inserts "item" at (optional)
            position "pos" in the list. The default value for "pos" is 0.

Examples:
>
        :let l = [1, 'item']
        :lua l = vim.eval('l') -- same 'l'
        :lua l:add(vim.list())
        :lua l[0] = math.pi
```

```
        :echo l[0] " 3.141593
        :lua l[0] = nil -- remove first item
        :lua l:insert(true, 1)
        :lua print(l, #l, l[0], l[1], l[-1])
        :lua for item in l() do print(item) end
<
```

==
4. Dict userdata *lua-dict*

Similarly to list userdata, dict userdata represent vim dictionaries; since dictionaries are also objects, references are kept between Lua and Vim. A dict "d" has the following properties:

Properties

- "#d" is the number of items in dict "d", equivalent to "len(d)" in Vim.
- "d.key" or "d['key']" returns the value at entry "key" in "d". To modify the entry at this key, simply do "d.key = newvalue"; in particular, "d.key = nil" removes the entry from "d".
- "d()" returns an iterator for "d" and is equivalent to "items(d)" in Vim.

Examples:

```
>
        :let d = {'n':10}
        :lua d = vim.eval('d') -- same 'd'
        :lua print(d, d.n, #d)
        :let d.self = d
        :lua for k, v in d() do print(d, k, v) end
        :lua d.x = math.pi
        :lua d.self = nil -- remove entry
        :echo d
<
```

==
5. Funcref userdata *lua-funcref*

Funcref userdata represent funcref variables in Vim. Funcrefs that were defined with a "dict" attribute need to be obtained as a dictionary key in order to have "self" properly assigned to the dictionary (see examples below.) A funcref "f" has the following properties:

Properties

- "#f" is the name of the function referenced by "f"
- "f(...)" calls the function referenced by "f" (with arguments)

Examples:

```
>
        :function I(x)
        :  return a:x
        :  endfunction
        :let R = function('I')
        :lua i1 = vim.funcref('I')
        :lua i2 = vim.eval('R')
        :lua print(#i1, #i2) -- both 'I'
        :lua print(i1, i2, #i2(i1) == #i1(i2))
        :function Mylen() dict
        :  return len(self.data)
        :  endfunction
```

```
        :let mydict = {'data': [0, 1, 2, 3]}
        :lua d = vim.eval('mydict'); d.len = vim.funcref('Mylen')
        :echo mydict.len()
        :lua l = d.len -- assign d as 'self'
        :lua print(l())
<
```

==

6. Buffer userdata *lua-buffer*

Buffer userdata represent vim buffers. A buffer userdata "b" has the following properties and methods:

Properties

- "b()" sets "b" as the current buffer.
- "#b" is the number of lines in buffer "b".
- "b[k]" represents line number k: "b[k] = newline" replaces line k with string "newline" and "b[k] = nil" deletes line k.
- "b.name" contains the short name of buffer "b" (read-only).
- "b.fname" contains the full name of buffer "b" (read-only).
- "b.number" contains the position of buffer "b" in the buffer list (read-only).

Methods

- "b:insert(newline[, pos])" inserts string "newline" at (optional) position "pos" in the buffer. The default value for "pos" is "#b + 1". If "pos == 0" then "newline" becomes the first line in the buffer.
- "b:next()" returns the buffer next to "b" in the buffer list.
- "b:previous()" returns the buffer previous to "b" in the buffer list.
- "b:isvalid()" returns 'true' (boolean) if buffer "b" corresponds to a "real" (not freed from memory) Vim buffer.

Examples:

```
>
        :lua b = vim.buffer() -- current buffer
        :lua print(b.name, b.number)
        :lua b[1] = "first line"
        :lua b:insert("FIRST!", 0)
        :lua b[1] = nil -- delete top line
        :lua for i=1,3 do b:insert(math.random()) end
        :3,4lua for i=vim.lastline,vim.firstline,-1 do b[i] = nil end
        :lua vim.open"myfile"() -- open buffer and set it as current

        function! ListBuffers()
        lua << EOF
        local b = vim.buffer(true) -- first buffer in list
        while b ~= nil do
                print(b.number, b.name, #b)
                b = b:next()
        end
        vim.beep()
        EOF
        endfunction
<
```

==

7. Window userdata *lua-window*

```
Window objects represent vim windows. A window userdata "w" has the following
properties and methods:

Properties
----------
        o "w()" sets "w" as the current window.
        o "w.buffer" contains the buffer of window "w" (read-only).
        o "w.line" represents the cursor line position in window "w".
        o "w.col" represents the cursor column position in window "w".
        o "w.width" represents the width of window "w".
        o "w.height" represents the height of window "w".

Methods
-------
        o "w:next()" returns the window next to "w".
        o "w:previous()" returns the window previous to "w".
        o "w:isvalid()" returns 'true' (boolean) if window "w" corresponds to
            a "real" (not freed from memory) Vim window.

Examples:
>
        :lua w = vim.window() -- current window
        :lua print(w.buffer.name, w.line, w.col)
        :lua w.width = w.width + math.random(10)
        :lua w.height = 2 * math.random() * w.height
        :lua n,w = 0,vim.window(true) while w~=nil do n,w = n + 1,w:next() end
        :lua print("There are " .. n .. " windows")
<

==============================================================================
8. The luaeval function                                 *lua-luaeval* *lua-eval*

The (dual) equivalent of "vim.eval" for passing Lua values to Vim is
"luaeval". "luaeval" takes an expression string and an optional argument and
returns the result of the expression. It is semantically equivalent in Lua to:
>
        local chunkheader = "local _A = select(1, ...) return "
        function luaeval (expstr, arg)
            local chunk = assert(loadstring(chunkheader .. expstr, "luaeval"))
            return chunk(arg) -- return typval
        end
<
Note that "_A" receives the argument to "luaeval". Lua numbers, strings, and
list, dict, and funcref userdata are converted to their Vim respective types,
while Lua booleans are converted to numbers. An error is thrown if conversion
of any of the remaining Lua types, including userdata other than lists, dicts,
and funcrefs, is attempted.

Examples: >

        :echo luaeval('math.pi')
        :lua a = vim.list():add('newlist')
        :let a = luaeval('a')
        :echo a[0] " 'newlist'
        :function Rand(x,y) " random uniform between x and y
        :  return luaeval('(_A.y-_A.x)*math.random()+_A.x', {'x':a:x,'y':a:y})
        :  endfunction
        :echo Rand(1,10)


==============================================================================
9. Dynamic loading                                      *lua-dynamic*
```

On MS-Windows and Unix the Lua library can be loaded dynamically. The
|:version| output then includes |+lua/dyn|.

This means that Vim will search for the Lua DLL or shared library file only
when needed. When you don't use the Lua interface you don't need it, thus
you can use Vim without this file.

MS-Windows ~

To use the Lua interface the Lua DLL must be in your search path. In a
console window type "path" to see what directories are used. The 'luadll'
option can be also used to specify the Lua DLL. The version of the DLL must
match the Lua version Vim was compiled with.

Unix ~

The 'luadll' option can be used to specify the Lua shared library file instead
of DYNAMIC_LUA_DLL file what was specified at compile time. The version of
the shared library must match the Lua version Vim was compiled with.

==
vim:tw=78:ts=8:noet:ft=help:norl:
if_mzsch.txt For Vim version 8.0. Last change: 2017 Oct 08

VIM REFERENCE MANUAL by Sergey Khorev

The MzScheme Interface to Vim *mzscheme* *MzScheme*

{Vi does not have any of these commands}

The MzScheme interface is available only if Vim was compiled with the
|+mzscheme| feature.

Based on the work of Brent Fulgham.
Dynamic loading added by Sergey Khorev

MzScheme and PLT Scheme names have been rebranded as Racket. For more
information please check http://racket-lang.org

Futures and places of Racket version 5.x up to and including 5.3.1 do not
work correctly with processes created by Vim.
The simplest solution is to build Racket on your own with these features
disabled: >
  ./configure --disable-futures --disable-places --prefix=your-install-prefix

To speed up the process, you might also want to use --disable-gracket and
--disable-docs

```
==============================================================================
1. Commands                                             *mzscheme-commands*

                                                        *:mzscheme* *:mz*
:[range]mz[scheme] {stmt}
                        Execute MzScheme statement {stmt}.  {not in Vi}

:[range]mz[scheme] << {endmarker}
{script}
{endmarker}
                        Execute inlined MzScheme script {script}.
                        Note: This command doesn't work if the MzScheme
                        feature wasn't compiled in.  To avoid errors, see
                        |script-here|.

                                                        *:mzfile* *:mzf*
:[range]mzf[ile] {file} Execute the MzScheme script in {file}.  {not in Vi}

All of these commands do essentially the same thing - they execute a piece of
MzScheme code, with the "current range" set to the given line
range.

In the case of :mzscheme, the code to execute is in the command-line.
In the case of :mzfile, the code to execute is the contents of the given file.

MzScheme interface defines exception exn:vim, derived from exn.
It is raised for various Vim errors.

During compilation, the MzScheme interface will remember the current MzScheme
collection path. If you want to specify additional paths use the
'current-library-collection-paths' parameter. E.g., to cons the user-local
MzScheme collection path: >
    :mz << EOF
    (current-library-collection-paths
        (cons
            (build-path (find-system-path 'addon-dir) (version) "collects")
            (current-library-collection-paths)))
    EOF
<

All functionality is provided through module vimext.

The exn:vim is available without explicit import.

To avoid clashes with MzScheme, consider using prefix when requiring module,
e.g.: >
        :mzscheme (require (prefix vim- vimext))
<
All the examples below assume this naming scheme.

                                                        *mzscheme-sandbox*
When executed in the |sandbox|, access to some filesystem and Vim interface
procedures is restricted.

==============================================================================
2. Examples                                             *mzscheme-examples*
>
        :mzscheme (display "Hello")
        :mz (display (string-append "Using MzScheme version " (version)))
        :mzscheme (require (prefix vim- vimext)) ; for MzScheme < 4.x
        :mzscheme (require (prefix-in vim- 'vimext)) ; MzScheme 4.x
```

```
        :mzscheme (vim-set-buff-line 10 "This is line #10")

To see what version of MzScheme you have: >
        :mzscheme (display (version))
<
Inline script usage: >
        function! <SID>SetFirstLine()
            :mz << EOF
            (display "!!!")
            (require (prefix vim- vimext))
            ; for newer versions (require (prefix-in vim- 'vimext))
            (vim-set-buff-line 1 "This is line #1")
            (vim-beep)
        EOF
        endfunction

        nmap <F9> :call <SID>SetFirstLine() <CR>
<
File execution: >
        :mzfile supascript.scm
<
Vim exception handling: >
        :mz << EOF
        (require (prefix vim- vimext))
        ; for newer versions (require (prefix-in vim- 'vimext))
        (with-handlers
          ([exn:vim? (lambda (e) (display (exn-message e)))])
          (vim-eval "nonsense-string"))
        EOF
<
Auto-instantiation of vimext module (can be placed in your |vimrc|): >
    function! MzRequire()
        :redir => l:mzversion
        :mz (version)
        :redir END
        if strpart(l:mzversion, 1, 1) < "4"
            " MzScheme versions < 4.x:
            :mz (require (prefix vim- vimext))
        else
            " newer versions:
            :mz (require (prefix-in vim- 'vimext))
        endif
    endfunction

    if has("mzscheme")
        silent call MzRequire()
    endif
<
==============================================================================
3. Threads                                              *mzscheme-threads*

The MzScheme interface supports threads. They are independent from OS threads,
thus scheduling is required. The option 'mzquantum' determines how often
Vim should poll for available MzScheme threads.
NOTE
Thread scheduling in the console version of Vim is less reliable than in the
GUI version.

==============================================================================
4. Vim access from MzScheme                             *mzscheme-vim*

                                                        *mzscheme-vimext*
```

```
The 'vimext' module provides access to procedures defined in the MzScheme
interface.

Common
------
    (command {command-string})      Perform the vim ":Ex" style command.
    (eval {expr-string})            Evaluate the vim expression into
                                    respective MzScheme object: |Lists| are
                                    represented as Scheme lists,
                                    |Dictionaries| as hash tables,
                                    |Funcref|s as functions (see also
                                    |mzscheme-funcref|)
                                    NOTE the name clashes with MzScheme eval,
                                    use module qualifiers to overcome this.
    (range-start)                   Start/End of the range passed with
    (range-end)                     the Scheme command.
    (beep)                          beep
    (get-option {option-name} [buffer-or-window]) Get Vim option value (either
                                    local or global, see set-option).
    (set-option {string} [buffer-or-window])
                                    Set a Vim option. String must have option
                                    setting form (like optname=optval, or
                                    optname+=optval, etc.) When called with
                                    {buffer} or {window} the local option will
                                    be set. The symbol 'global can be passed
                                    as {buffer-or-window}. Then |:setglobal|
                                    will be used.

Buffers                                                 *mzscheme-buffer*
-------
    (buff? {object})                Is object a buffer?
    (buff-valid? {object})          Is object a valid buffer? (i.e.
                                    corresponds to the real Vim buffer)
    (get-buff-line {linenr} [buffer])
                                    Get line from a buffer.
    (set-buff-line {linenr} {string} [buffer])
                                    Set a line in a buffer. If {string} is #f,
                                    the line gets deleted.  The [buffer]
                                    argument is optional. If omitted, the
                                    current buffer will be used.
    (get-buff-line-list {start} {end} [buffer])
                                    Get a list of lines in a buffer. {Start}
                                    and {end} are 1-based and inclusive.
    (set-buff-line-list {start} {end} {string-list} [buffer])
                                    Set a list of lines in a buffer. If
                                    string-list is #f or null, the lines get
                                    deleted. If a list is shorter than
                                    {end}-{start} the remaining lines will
                                    be deleted.
    (get-buff-name [buffer])        Get a buffer's text name.
    (get-buff-num [buffer])         Get a buffer's number.
    (get-buff-size [buffer])        Get buffer line count.
    (insert-buff-line-list {linenr} {string/string-list} [buffer])
                                    Insert a list of lines into a buffer after
                                    {linenr}. If {linenr} is 0, lines will be
                                    inserted at start.
    (curr-buff)                     Get the current buffer. Use other MzScheme
                                    interface procedures to change it.
    (buff-count)                    Get count of total buffers in the editor.
    (get-next-buff [buffer])        Get next buffer.
    (get-prev-buff [buffer])        Get previous buffer. Return #f when there
                                    are no more buffers.
```

```
    (open-buff {filename})          Open a new buffer (for file "name")
    (get-buff-by-name {buffername}) Get a buffer by its filename or #f
                                        if there is no such buffer.
    (get-buff-by-num {buffernum})   Get a buffer by its number (return #f if
                                    there is no buffer with this number).

Windows                                                     *mzscheme-window*
------
    (win? {object})                 Is object a window?
    (win-valid? {object})           Is object a valid window (i.e. corresponds
                                    to the real Vim window)?
    (curr-win)                      Get the current window.
    (win-count)                     Get count of windows.
    (get-win-num [window])          Get window number.
    (get-win-by-num {windownum})    Get window by its number.
    (get-win-buffer     [window])   Get the buffer for a given window.
    (get-win-height [window])
    (set-win-height {height} [window])  Get/Set height of window.
    (get-win-width [window])
    (set-win-width {width} [window])Get/Set width of window.
    (get-win-list [buffer])         Get list of windows for a buffer.
    (get-cursor [window])           Get cursor position in a window as
                                    a pair (linenr . column).
    (set-cursor (line . col) [window])  Set cursor position.

==============================================================================
5. mzeval() Vim function                                    *mzscheme-mzeval*

To facilitate bi-directional interface, you can use |mzeval()| function to
evaluate MzScheme expressions and pass their values to Vim script.

==============================================================================
6. Using Function references                               *mzscheme-funcref*

MzScheme interface allows use of |Funcref|s so you can call Vim functions
directly from Scheme. For instance: >
    function! MyAdd2(arg)
        return a:arg + 2
    endfunction
    mz (define f2 (vim-eval "function(\"MyAdd2\")"))
    mz (f2 7)
< or : >
    :mz (define indent (vim-eval "function('indent')"))
    " return Vim indent for line 12
    :mz (indent 12)
<

==============================================================================
7. Dynamic loading                              *mzscheme-dynamic* *E815*

On MS-Windows the MzScheme libraries can be loaded dynamically. The |:version|
output then includes |+mzscheme/dyn|.

This means that Vim will search for the MzScheme DLL files only when needed.
When you don't use the MzScheme interface you don't need them, thus you can
use Vim without these DLL files.
NOTE: Newer version of MzScheme (Racket) require earlier (trampolined)
initialisation via scheme_main_setup.  So Vim always loads the MzScheme DLL at
startup if possible.  This may make Vim startup slower.

To use the MzScheme interface the MzScheme DLLs must be in your search path.
In a console window type "path" to see what directories are used.
```

```
On MS-Windows the options 'mzschemedll' and 'mzschemegcdll' are used for the
name of the library to load.  The initial value is specified at build time.

The version of the DLL must match the MzScheme version Vim was compiled with.
For MzScheme version 209 they will be "libmzsch209_000.dll" and
"libmzgc209_000.dll". To know for sure look at the output of the ":version"
command, look for -DDYNAMIC_MZSCH_DLL="something" and
-DDYNAMIC_MZGC_DLL="something" in the "Compilation" info.

For example, if MzScheme (Racket) is installed at C:\Racket63, you may need
to set the environment variable as the following: >

  PATH=%PATH%;C:\Racket63\lib
  PLTCOLLECTS=C:\Racket63\collects
  PLTCONFIGDIR=C:\Racket63\etc
<
==============================================================================
8. MzScheme setup                                       *mzscheme-setup* *E895*

Vim requires "racket/base" module for if_mzsch core (fallback to "scheme/base"
if it doesn't exist), "r5rs" module for test and "raco ctool" command for
building Vim.  If MzScheme did not have them, you can install them with
MzScheme's raco command:
>
  raco pkg install scheme-lib       # scheme/base module
  raco pkg install r5rs-lib         # r5rs module
  raco pkg install cext-lib         # raco ctool command
<
==============================================================================
  vim:tw=78:ts=8:sts=4:ft=help:norl:
*if_perl.txt*   For Vim version 8.0.  Last change: 2015 Oct 16


                  VIM REFERENCE MANUAL    by Sven Verdoolaege
                                         and Matt Gerassimof

Perl and Vim                             *perl* *Perl*

1. Editing Perl files                    |perl-editing|
2. Compiling Vim with Perl interface     |perl-compiling|
3. Using the Perl interface              |perl-using|
4. Dynamic loading                       |perl-dynamic|

{Vi does not have any of these commands}

The Perl interface only works when Vim was compiled with the |+perl| feature.

==============================================================================
1. Editing Perl files                                    *perl-editing*

Vim syntax highlighting supports Perl and POD files.  Vim assumes a file is
Perl code if the filename has a .pl or .pm suffix.  Vim also examines the first
line of a file, regardless of the filename suffix, to check if a file is a
Perl script (see scripts.vim in Vim's syntax directory).  Vim assumes a file
is POD text if the filename has a .POD suffix.

To use tags with Perl, you need a recent version of Exuberant ctags.  Look
here:
        http://ctags.sourceforge.net

Alternatively, you can use the Perl script pltags.pl, which is shipped with
```

```
Vim in the $VIMRUNTIME/tools directory.  This script has currently more
features than Exuberant ctags' Perl support.

==============================================================================
2. Compiling Vim with Perl interface                    *perl-compiling*

To compile Vim with Perl interface, you need Perl 5.004 (or later).  Perl must
be installed before you compile Vim.  Vim's Perl interface does NOT work with
the 5.003 version that has been officially released!  It will probably work
with Perl 5.003_05 and later.

The Perl patches for Vim were made by:
        Sven Verdoolaege <skimo@breughel.ufsia.ac.be>
        Matt Gerassimof

Perl for MS-Windows can be found at: http://www.perl.com/
The ActiveState one should work.

==============================================================================
3. Using the Perl interface                             *perl-using*

                                                        *:perl* *:pe*
:pe[rl] {cmd}           Execute Perl command {cmd}.  The current package
                        is "main".  Simple example to test if `:perl` is
                        working: >
                                :perl VIM::Msg("Hello")

:pe[rl] << {endpattern}
{script}
{endpattern}
                        Execute Perl script {script}.
                        {endpattern} must NOT be preceded by any white space.
                        If {endpattern} is omitted, it defaults to a dot '.'
                        like for the |:append| and |:insert| commands.  Using
                        '.' helps when inside a function, because "$i;" looks
                        like the start of an |:insert| command to Vim.
                        This form of the |:perl| command is mainly useful for
                        including perl code in vim scripts.
                        Note: This command doesn't work when the Perl feature
                        wasn't compiled in.  To avoid errors, see
                        |script-here|.


Example vim script: >

        function! WhitePearl()
        perl << EOF
                VIM::Msg("pearls are nice for necklaces");
                VIM::Msg("rubys for rings");
                VIM::Msg("pythons for bags");
                VIM::Msg("tcls????");
        EOF
        endfunction
<
To see what version of Perl you have: >
        :perl print $^V
<

                                                        *:perldo* *:perld*
:[range]perld[o] {cmd}  Execute Perl command {cmd} for each line in the
                        [range], with $_ being set to the text of each line in
                        turn, without a trailing <EOL>.  Setting $_ will change
```

```
                        the text, but note that it is not possible to add or
                        delete lines using this command.
                        The default for [range] is the whole file: "1,$".

Here are some things you can try: >

  :perl $a=1
  :perldo $_ = reverse($_);1
  :perl VIM::Msg("hello")
  :perl $line = $curbuf->Get(42)
<
                                                        *E299*
Executing Perl commands in the |sandbox| is limited.  ":perldo" will not be
possible at all.  ":perl" will be evaluated in the Safe environment, if
possible.


                                                        *perl-overview*
Here is an overview of the functions that are available to Perl: >

  :perl VIM::Msg("Text")               # displays a message
  :perl VIM::Msg("Error", "ErrorMsg")   # displays an error message
  :perl VIM::Msg("remark", "Comment")   # displays a highlighted message
  :perl VIM::SetOption("ai")           # sets a vim option
  :perl $nbuf = VIM::Buffers()         # returns the number of buffers
  :perl @buflist = VIM::Buffers()      # returns array of all buffers
  :perl $mybuf = (VIM::Buffers('qq.c'))[0] # returns buffer object for 'qq.c'
  :perl @winlist = VIM::Windows()      # returns array of all windows
  :perl $nwin = VIM::Windows()         # returns the number of windows
  :perl ($success, $v) = VIM::Eval('&path') # $v: option 'path', $success: 1
  :perl ($success, $v) = VIM::Eval('&xyz')  # $v: '' and $success: 0
  :perl $v = VIM::Eval('expand("<cfile>")') # expands <cfile>
  :perl $curwin->SetHeight(10)         # sets the window height
  :perl @pos = $curwin->Cursor()       # returns (row, col) array
  :perl @pos = (10, 10)
  :perl $curwin->Cursor(@pos)          # sets cursor to @pos
  :perl $curwin->Cursor(10,10)         # sets cursor to row 10 col 10
  :perl $mybuf = $curwin->Buffer()     # returns the buffer object for window
  :perl $curbuf->Name()                # returns buffer name
  :perl $curbuf->Number()              # returns buffer number
  :perl $curbuf->Count()               # returns the number of lines
  :perl $l = $curbuf->Get(10)          # returns line 10
  :perl @l = $curbuf->Get(1 .. 5)      # returns lines 1 through 5
  :perl $curbuf->Delete(10)            # deletes line 10
  :perl $curbuf->Delete(10, 20)        # delete lines 10 through 20
  :perl $curbuf->Append(10, "Line")    # appends a line
  :perl $curbuf->Append(10, "Line1", "Line2", "Line3") # appends 3 lines
  :perl @l = ("L1", "L2", "L3")
  :perl $curbuf->Append(10, @l)        # appends L1, L2 and L3
  :perl $curbuf->Set(10, "Line")       # replaces line 10
  :perl $curbuf->Set(10, "Line1", "Line2")     # replaces lines 10 and 11
  :perl $curbuf->Set(10, @l)           # replaces 3 lines
<
                                                        *perl-Msg*
VIM::Msg({msg}, {group}?)
                        Displays the message {msg}.  The optional {group}
                        argument specifies a highlight group for Vim to use
                        for the message.

                                                        *perl-SetOption*
VIM::SetOption({arg})   Sets a vim option.  {arg} can be any argument that the
                        ":set" command accepts.  Note that this means that no
```

```
                        spaces are allowed in the argument!  See |:set|.

                                                        *perl-Buffers*
VIM::Buffers([{bn}...]) With no arguments, returns a list of all the buffers
                        in an array context or returns the number of buffers
                        in a scalar context.  For a list of buffer names or
                        numbers {bn}, returns a list of the buffers matching
                        {bn}, using the same rules as Vim's internal
                        |bufname()| function.
                        WARNING: the list becomes invalid when |:bwipe| is
                        used.  Using it anyway may crash Vim.

                                                        *perl-Windows*
VIM::Windows([{wn}...]) With no arguments, returns a list of all the windows
                        in an array context or returns the number of windows
                        in a scalar context.  For a list of window numbers
                        {wn}, returns a list of the windows with those
                        numbers.
                        WARNING: the list becomes invalid when a window is
                        closed.  Using it anyway may crash Vim.

                                                        *perl-DoCommand*
VIM::DoCommand({cmd})   Executes Ex command {cmd}.

                                                        *perl-Eval*
VIM::Eval({expr})       Evaluates {expr} and returns (success, value) in list
                        context or just value in scalar context.
                        success=1 indicates that val contains the value of
                        {expr}; success=0 indicates a failure to evaluate
                        the expression.  '@x' returns the contents of register
                        x, '&x' returns the value of option x, 'x' returns the
                        value of internal |variables| x, and '$x' is equivalent
                        to perl's $ENV{x}.  All |functions| accessible from
                        the command-line are valid for {expr}.
                        A |List| is turned into a string by joining the items
                        and inserting line breaks.

                                                        *perl-SetHeight*
Window->SetHeight({height})
                        Sets the Window height to {height}, within screen
                        limits.

                                                        *perl-GetCursor*
Window->Cursor({row}?, {col}?)
                        With no arguments, returns a (row, col) array for the
                        current cursor position in the Window.  With {row} and
                        {col} arguments, sets the Window's cursor position to
                        {row} and {col}.  Note that {col} is numbered from 0,
                        Perl-fashion, and thus is one less than the value in
                        Vim's ruler.

Window->Buffer()                                        *perl-Buffer*
                        Returns the Buffer object corresponding to the given
                        Window.

                                                        *perl-Name*
Buffer->Name()          Returns the filename for the Buffer.

                                                        *perl-Number*
Buffer->Number()        Returns the number of the Buffer.

                                                        *perl-Count*
```

```
Buffer->Count()         Returns the number of lines in the Buffer.

                                                        *perl-Get*
Buffer->Get({lnum}, {lnum}?, ...)
                        Returns a text string of line {lnum} in the Buffer
                        for each {lnum} specified.  An array can be passed
                        with a list of {lnum}'s specified.

                                                        *perl-Delete*
Buffer->Delete({lnum}, {lnum}?)
                        Deletes line {lnum} in the Buffer.  With the second
                        {lnum}, deletes the range of lines from the first
                        {lnum} to the second {lnum}.

                                                        *perl-Append*
Buffer->Append({lnum}, {line}, {line}?, ...)
                        Appends each {line} string after Buffer line {lnum}.
                        The list of {line}s can be an array.

                                                        *perl-Set*
Buffer->Set({lnum}, {line}, {line}?, ...)
                        Replaces one or more Buffer lines with specified
                        {lines}s, starting at Buffer line {lnum}.  The list of
                        {line}s can be an array.  If the arguments are
                        invalid, replacement does not occur.

$main::curwin
                        The current window object.

$main::curbuf
                        The current buffer object.


                                                        *script-here*
When using a script language in-line, you might want to skip this when the
language isn't supported.  But this mechanism doesn't work: >
   if has('perl')
     perl << EOF
       this will NOT work!
   EOF
   endif
Instead, put the Perl/Python/Ruby/etc. command in a function and call that
function: >
    if has('perl')
      function DefPerl()
        perl << EOF
          this works
    EOF
      endfunction
      call DefPerl()
    endif
Note that "EOF" must be at the start of the line.

==============================================================================
4. Dynamic loading                                      *perl-dynamic*

On MS-Windows and Unix the Perl library can be loaded dynamically.  The
|:version| output then includes |+perl/dyn|.

This means that Vim will search for the Perl DLL or shared library file only
when needed.  When you don't use the Perl interface you don't need it, thus
you can use Vim without this file.
```

MS-Windows ~

You can download Perl from http://www.perl.org. The one from ActiveState was
used for building Vim.

To use the Perl interface the Perl DLL must be in your search path.
If Vim reports it cannot find the perl512.dll, make sure your $PATH includes
the directory where it is located. The Perl installer normally does that.
In a console window type "path" to see what directories are used. The
'perldll' option can be also used to specify the Perl DLL.

The name of the DLL must match the Perl version Vim was compiled with.
Currently the name is "perl512.dll". That is for Perl 5.12. To know for
sure edit "gvim.exe" and search for "perl\d*.dll\c".

Unix ~

The 'perldll' option can be used to specify the Perl shared library file
instead of DYNAMIC_PERL_DLL file what was specified at compile time. The
version of the shared library must match the Perl version Vim was compiled
with.

==
 vim:tw=78:ts=8:ft=help:norl:
if_pyth.txt   For Vim version 8.0. Last change: 2017 Mar 09

                  VIM REFERENCE MANUAL   by Paul Moore

The Python Interface to Vim   *python* *Python*

1. Commands |python-commands|
2. The vim module |python-vim|
3. Buffer objects |python-buffer|
4. Range objects |python-range|
5. Window objects |python-window|
6. Tab page objects |python-tabpage|
7. vim.bindeval objects |python-bindeval-objects|
8. pyeval(), py3eval() Vim functions |python-pyeval|
9. Dynamic loading |python-dynamic|
10. Python 3 |python3|
11. Python X |python_x|
12. Building with Python support |python-building|

{Vi does not have any of these commands}

The Python 2.x interface is available only when Vim was compiled with the
|+python| feature.
The Python 3 interface is available only when Vim was compiled with the
|+python3| feature.
Both can be available at the same time, but read |python-2-and-3|.

==
1. Commands   *python-commands*

                                        :python *:py* *E263* *E264* *E887*
:[range]py[thon] {stmt}
                        Execute Python statement {stmt}. A simple check if

```
                        the `:python` command is working: >
                                :python print "Hello"

:[range]py[thon] << {endmarker}
{script}
{endmarker}
                        Execute Python script {script}.
                        Note: This command doesn't work when the Python
                        feature wasn't compiled in.  To avoid errors, see
                        |script-here|.

{endmarker} must NOT be preceded by any white space.  If {endmarker} is
omitted from after the "<<", a dot '.' must be used after {script}, like
for the |:append| and |:insert| commands.
This form of the |:python| command is mainly useful for including python code
in Vim scripts.

Example: >
        function! IcecreamInitialize()
        python << EOF
        class StrawberryIcecream:
                def __call__(self):
                        print 'EAT ME'
        EOF
        endfunction

To see what version of Python you have: >
        :python import sys
        :python print(sys.version)

Note: Python is very sensitive to the indenting.  Make sure the "class" line
and "EOF" do not have any indent.

                                                        *:pydo*
:[range]pydo {body}     Execute Python function "def _vim_pydo(line, linenr):
                        {body}" for each line in the [range], with the
                        function arguments being set to the text of each line
                        in turn, without a trailing <EOL>, and the current
                        line number. The function should return a string or
                        None. If a string is returned, it becomes the text of
                        the line in the current turn. The default for [range]
                        is the whole file: "1,$".
                        {not in Vi}

Examples:
>
        :pydo return "%s\t%d" % (line[::-1], len(line))
        :pydo if line: return "%4d: %s" % (linenr, line)
<
                                                        *:pyfile* *:pyf*
:[range]pyf[ile] {file}
                        Execute the Python script in {file}.  The whole
                        argument is used as a single file name.  {not in Vi}

Both of these commands do essentially the same thing - they execute a piece of
Python code, with the "current range" |python-range| set to the given line
range.

In the case of :python, the code to execute is in the command-line.
In the case of :pyfile, the code to execute is the contents of the given file.

Python commands cannot be used in the |sandbox|.
```

```
To pass arguments you need to set sys.argv[] explicitly.  Example: >

        :python import sys
        :python sys.argv = ["foo", "bar"]
        :pyfile myscript.py

Here are some examples                                  *python-examples*  >

        :python from vim import *
        :python from string import upper
        :python current.line = upper(current.line)
        :python print "Hello"
        :python str = current.buffer[42]

(Note that changes - like the imports - persist from one command to the next,
just like in the Python interpreter.)

==============================================================================
2. The vim module                                       *python-vim*

Python code gets all of its access to vim (with one exception - see
|python-output| below) via the "vim" module.  The vim module implements two
methods, three constants, and one error object.  You need to import the vim
module before using it: >
        :python import vim

Overview >
        :py print "Hello"               # displays a message
        :py vim.command(cmd)            # execute an Ex command
        :py w = vim.windows[n]          # gets window "n"
        :py cw = vim.current.window     # gets the current window
        :py b = vim.buffers[n]          # gets buffer "n"
        :py cb = vim.current.buffer     # gets the current buffer
        :py w.height = lines            # sets the window height
        :py w.cursor = (row, col)       # sets the window cursor position
        :py pos = w.cursor              # gets a tuple (row, col)
        :py name = b.name               # gets the buffer file name
        :py line = b[n]                 # gets a line from the buffer
        :py lines = b[n:m]              # gets a list of lines
        :py num = len(b)                # gets the number of lines
        :py b[n] = str                  # sets a line in the buffer
        :py b[n:m] = [str1, str2, str3] # sets a number of lines at once
        :py del b[n]                    # deletes a line
        :py del b[n:m]                  # deletes a number of lines


Methods of the "vim" module

vim.command(str)                                        *python-command*
        Executes the vim (ex-mode) command str.  Returns None.
        Examples: >
            :py vim.command("set tw=72")
            :py vim.command("%s/aaa/bbb/g")
<       The following definition executes Normal mode commands: >
                def normal(str):
                        vim.command("normal "+str)
                # Note the use of single quotes to delimit a string containing
                # double quotes
                normal('"a2dd"aP')
<                                                               *E659*
        The ":python" command cannot be used recursively with Python 2.2 and
```

```
        older.  This only works with Python 2.3 and later: >
            :py vim.command("python print 'Hello again Python'")

vim.eval(str)                                           *python-eval*
        Evaluates the expression str using the vim internal expression
        evaluator (see |expression|).  Returns the expression result as:
        - a string if the Vim expression evaluates to a string or number
        - a list if the Vim expression evaluates to a Vim list
        - a dictionary if the Vim expression evaluates to a Vim dictionary
        Dictionaries and lists are recursively expanded.
        Examples: >
            :py text_width = vim.eval("&tw")
            :py str = vim.eval("12+12")         # NB result is a string! Use
                                                # string.atoi() to convert to
                                                # a number.

            :py tagList = vim.eval('taglist("eval_expr")')
<       The latter will return a python list of python dicts, for instance:
        [{'cmd': '/^eval_expr(arg, nextcmd)$/', 'static': 0, 'name': ~
        'eval_expr', 'kind': 'f', 'filename': './src/eval.c'}] ~

vim.bindeval(str)                                       *python-bindeval*
        Like |python-eval|, but returns special objects described in
        |python-bindeval-objects|. These python objects let you modify (|List|
        or |Dictionary|) or call (|Funcref|) vim objects.

vim.strwidth(str)                                       *python-strwidth*
        Like |strwidth()|: returns number of display cells str occupies, tab
        is counted as one cell.

vim.foreach_rtp(callable)                               *python-foreach_rtp*
        Call the given callable for each path in 'runtimepath' until either
        callable returns something but None, the exception is raised or there
        are no longer paths. If stopped in case callable returned non-None,
        vim.foreach_rtp function returns the value returned by callable.

vim.chdir(*args, **kwargs)                              *python-chdir*
vim.fchdir(*args, **kwargs)                             *python-fchdir*
        Run os.chdir or os.fchdir, then all appropriate vim stuff.
        Note: you should not use these functions directly, use os.chdir and
              os.fchdir instead. Behavior of vim.fchdir is undefined in case
              os.fchdir does not exist.

Error object of the "vim" module

vim.error                                               *python-error*
        Upon encountering a Vim error, Python raises an exception of type
        vim.error.
        Example: >
                try:
                        vim.command("put a")
                except vim.error:
                        # nothing in register a

Constants of the "vim" module

        Note that these are not actually constants - you could reassign them.
        But this is silly, as you would then lose access to the vim objects
        to which the variables referred.

vim.buffers                                             *python-buffers*
        A mapping object providing access to the list of vim buffers.  The
```

```
        object supports the following operations: >
            :py b = vim.buffers[i]      # Indexing (read-only)
            :py b in vim.buffers        # Membership test
            :py n = len(vim.buffers)    # Number of elements
            :py for b in vim.buffers:   # Iterating over buffer list
<
vim.windows                                             *python-windows*
        A sequence object providing access to the list of vim windows.  The
        object supports the following operations: >
            :py w = vim.windows[i]      # Indexing (read-only)
            :py w in vim.windows        # Membership test
            :py n = len(vim.windows)    # Number of elements
            :py for w in vim.windows:   # Sequential access
<       Note: vim.windows object always accesses current tab page.
        |python-tabpage|.windows objects are bound to parent |python-tabpage|
        object and always use windows from that tab page (or throw vim.error
        in case tab page was deleted). You can keep a reference to both
        without keeping a reference to vim module object or |python-tabpage|,
        they will not lose their properties in this case.

vim.tabpages                                            *python-tabpages*
        A sequence object providing access to the list of vim tab pages. The
        object supports the following operations: >
            :py t = vim.tabpages[i]     # Indexing (read-only)
            :py t in vim.tabpages       # Membership test
            :py n = len(vim.tabpages)   # Number of elements
            :py for t in vim.tabpages:  # Sequential access
<
vim.current                                             *python-current*
        An object providing access (via specific attributes) to various
        "current" objects available in vim:
                vim.current.line        The current line (RW)           String
                vim.current.buffer      The current buffer (RW)         Buffer
                vim.current.window      The current window (RW)         Window
                vim.current.tabpage     The current tab page (RW)       TabPage
                vim.current.range       The current line range (RO)     Range

        The last case deserves a little explanation.  When the :python or
        :pyfile command specifies a range, this range of lines becomes the
        "current range".  A range is a bit like a buffer, but with all access
        restricted to a subset of lines.  See |python-range| for more details.

        Note: When assigning to vim.current.{buffer,window,tabpage} it expects
        valid |python-buffer|, |python-window| or |python-tabpage| objects
        respectively. Assigning triggers normal (with |autocommand|s)
        switching to given buffer, window or tab page. It is the only way to
        switch UI objects in python: you can't assign to
        |python-tabpage|.window attribute. To switch without triggering
        autocommands use >
            py << EOF
            saved_eventignore = vim.options['eventignore']
            vim.options['eventignore'] = 'all'
            try:
                vim.current.buffer = vim.buffers[2] # Switch to buffer 2
            finally:
                vim.options['eventignore'] = saved_eventignore
            EOF
<
vim.vars                                                *python-vars*
vim.vvars                                               *python-vvars*
        Dictionary-like objects holding dictionaries with global (|g:|) and
        vim (|v:|) variables respectively. Identical to `vim.bindeval("g:")`,
```

```
        but faster.

vim.options                                              *python-options*
        Object partly supporting mapping protocol (supports setting and
        getting items) providing a read-write access to global options.
        Note: unlike |:set| this provides access only to global options. You
        cannot use this object to obtain or set local options' values or
        access local-only options in any fashion. Raises KeyError if no global
        option with such name exists (i.e. does not raise KeyError for
        |global-local| options and global only options, but does for window-
        and buffer-local ones).  Use |python-buffer| objects to access to
        buffer-local options and |python-window| objects to access to
        window-local options.

        Type of this object is available via "Options" attribute of vim
        module.

Output from Python                                       *python-output*
        Vim displays all Python code output in the Vim message area.  Normal
        output appears as information messages, and error output appears as
        error messages.

        In implementation terms, this means that all output to sys.stdout
        (including the output from print statements) appears as information
        messages, and all output to sys.stderr (including error tracebacks)
        appears as error messages.

                                                         *python-input*
        Input (via sys.stdin, including input() and raw_input()) is not
        supported, and may cause the program to crash.  This should probably be
        fixed.

                    *python2-directory* *python3-directory* *pythonx-directory*
Python 'runtimepath' handling                            *python-special-path*

In python vim.VIM_SPECIAL_PATH special directory is used as a replacement for
the list of paths found in 'runtimepath': with this directory in sys.path and
vim.path_hooks in sys.path_hooks python will try to load module from
{rtp}/python2 (or python3) and {rtp}/pythonx (for both python versions) for
each {rtp} found in 'runtimepath'.

Implementation is similar to the following, but written in C: >

    from imp import find_module, load_module
    import vim
    import sys

    class VimModuleLoader(object):
        def __init__(self, module):
            self.module = module

        def load_module(self, fullname, path=None):
            return self.module

    def _find_module(fullname, oldtail, path):
        idx = oldtail.find('.')
        if idx > 0:
            name = oldtail[:idx]
            tail = oldtail[idx+1:]
            fmr = find_module(name, path)
            module = load_module(fullname[:-len(oldtail)] + name, *fmr)
            return _find_module(fullname, tail, module.__path__)
```

```
        else:
            fmr = find_module(fullname, path)
            return load_module(fullname, *fmr)

    # It uses vim module itself in place of VimPathFinder class: it does not
    # matter for python which object has find_module function attached to as
    # an attribute.
    class VimPathFinder(object):
        @classmethod
        def find_module(cls, fullname, path=None):
            try:
                return VimModuleLoader(_find_module(fullname, fullname, path or
vim._get_paths()))
            except ImportError:
                return None

        @classmethod
        def load_module(cls, fullname, path=None):
            return _find_module(fullname, fullname, path or vim._get_paths())

    def hook(path):
        if path == vim.VIM_SPECIAL_PATH:
            return VimPathFinder
        else:
            raise ImportError

    sys.path_hooks.append(hook)

vim.VIM_SPECIAL_PATH                                    *python-VIM_SPECIAL_PATH*
        String constant used in conjunction with vim path hook. If path hook
        installed by vim is requested to handle anything but path equal to
        vim.VIM_SPECIAL_PATH constant it raises ImportError. In the only other
        case it uses special loader.

        Note: you must not use value of this constant directly, always use
              vim.VIM_SPECIAL_PATH object.

vim.find_module(...)                                    *python-find_module*
vim.path_hook(path)                                     *python-path_hook*
        Methods or objects used to implement path loading as described above.
        You should not be using any of these directly except for vim.path_hook
        in case you need to do something with sys.meta_path. It is not
        guaranteed that any of the objects will exist in the future vim
        versions.

vim._get_paths                                          *python-_get_paths*
        Methods returning a list of paths which will be searched for by path
        hook. You should not rely on this method being present in future
        versions, but can use it for debugging.

        It returns a list of {rtp}/python2 (or {rtp}/python3) and
        {rtp}/pythonx directories for each {rtp} in 'runtimepath'.

==============================================================================
3. Buffer objects                                       *python-buffer*

Buffer objects represent vim buffers.  You can obtain them in a number of ways:
        - via vim.current.buffer (|python-current|)
        - from indexing vim.buffers (|python-buffers|)
        - from the "buffer" attribute of a window (|python-window|)

Buffer objects have two read-only attributes - name - the full file name for
```

```
the buffer, and number - the buffer number.  They also have three methods
(append, mark, and range; see below).

You can also treat buffer objects as sequence objects.  In this context, they
act as if they were lists (yes, they are mutable) of strings, with each
element being a line of the buffer.  All of the usual sequence operations,
including indexing, index assignment, slicing and slice assignment, work as
you would expect.  Note that the result of indexing (slicing) a buffer is a
string (list of strings).  This has one unusual consequence - b[:] is different
from b.  In particular, "b[:] = None" deletes the whole of the buffer, whereas
"b = None" merely updates the variable b, with no effect on the buffer.

Buffer indexes start at zero, as is normal in Python.  This differs from vim
line numbers, which start from 1.  This is particularly relevant when dealing
with marks (see below) which use vim line numbers.

The buffer object attributes are:
        b.vars          Dictionary-like object used to access
                        |buffer-variable|s.
        b.options       Mapping object (supports item getting, setting and
                        deleting) that provides access to buffer-local options
                        and buffer-local values of |global-local| options. Use
                        |python-window|.options if option is window-local,
                        this object will raise KeyError. If option is
                        |global-local| and local value is missing getting it
                        will return None.
        b.name          String, RW. Contains buffer name (full path).
                        Note: when assigning to b.name |BufFilePre| and
                        |BufFilePost| autocommands are launched.
        b.number        Buffer number. Can be used as |python-buffers| key.
                        Read-only.
        b.valid         True or False. Buffer object becomes invalid when
                        corresponding buffer is wiped out.

The buffer object methods are:
        b.append(str)   Append a line to the buffer
        b.append(str, nr)  Idem, below line "nr"
        b.append(list)  Append a list of lines to the buffer
                        Note that the option of supplying a list of strings to
                        the append method differs from the equivalent method
                        for Python's built-in list objects.
        b.append(list, nr)  Idem, below line "nr"
        b.mark(name)    Return a tuple (row,col) representing the position
                        of the named mark (can also get the []"<> marks)
        b.range(s,e)    Return a range object (see |python-range|) which
                        represents the part of the given buffer between line
                        numbers s and e |inclusive|.

Note that when adding a line it must not contain a line break character '\n'.
A trailing '\n' is allowed and ignored, so that you can do: >
        :py b.append(f.readlines())

Buffer object type is available using "Buffer" attribute of vim module.

Examples (assume b is the current buffer) >
        :py print b.name               # write the buffer file name
        :py b[0] = "hello!!!"          # replace the top line
        :py b[:] = None                # delete the whole buffer
        :py del b[:]                   # delete the whole buffer
        :py b[0:0] = [ "a line" ]      # add a line at the top
        :py del b[2]                   # delete a line (the third)
        :py b.append("bottom")         # add a line at the bottom
```

```
        :py n = len(b)                  # number of lines
        :py (row,col) = b.mark('a')     # named mark
        :py r = b.range(1,5)            # a sub-range of the buffer
        :py b.vars["foo"] = "bar"       # assign b:foo variable
        :py b.options["ff"] = "dos"     # set fileformat
        :py del b.options["ar"]         # same as :set autoread<

==============================================================================
4. Range objects                                        *python-range*

Range objects represent a part of a vim buffer.  You can obtain them in a
number of ways:
        - via vim.current.range (|python-current|)
        - from a buffer's range() method (|python-buffer|)

A range object is almost identical in operation to a buffer object.  However,
all operations are restricted to the lines within the range (this line range
can, of course, change as a result of slice assignments, line deletions, or
the range.append() method).

The range object attributes are:
        r.start         Index of first line into the buffer
        r.end           Index of last line into the buffer

The range object methods are:
        r.append(str)   Append a line to the range
        r.append(str, nr)  Idem, after line "nr"
        r.append(list)  Append a list of lines to the range
                        Note that the option of supplying a list of strings to
                        the append method differs from the equivalent method
                        for Python's built-in list objects.
        r.append(list, nr)  Idem, after line "nr"

Range object type is available using "Range" attribute of vim module.

Example (assume r is the current range):
        # Send all lines in a range to the default printer
        vim.command("%d,%dhardcopy!" % (r.start+1,r.end+1))

==============================================================================
5. Window objects                                       *python-window*

Window objects represent vim windows.  You can obtain them in a number of ways:
        - via vim.current.window (|python-current|)
        - from indexing vim.windows (|python-windows|)
        - from indexing "windows" attribute of a tab page (|python-tabpage|)
        - from the "window" attribute of a tab page (|python-tabpage|)

You can manipulate window objects only through their attributes.  They have no
methods, and no sequence or other interface.

Window attributes are:
        buffer (read-only)      The buffer displayed in this window
        cursor (read-write)     The current cursor position in the window
                                This is a tuple, (row,col).
        height (read-write)     The window height, in rows
        width (read-write)      The window width, in columns
        vars (read-only)        The window |w:| variables. Attribute is
                                unassignable, but you can change window
                                variables this way
        options (read-only)     The window-local options. Attribute is
                                unassignable, but you can change window
```

```
                                options this way. Provides access only to
                                window-local options, for buffer-local use
                                |python-buffer| and for global ones use
                                |python-options|. If option is |global-local|
                                and local value is missing getting it will
                                return None.
        number (read-only)      Window number.  The first window has number 1.
                                This is zero in case it cannot be determined
                                (e.g. when the window object belongs to other
                                tab page).
        row, col (read-only)    On-screen window position in display cells.
                                First position is zero.
        tabpage (read-only)     Window tab page.
        valid (read-write)      True or False. Window object becomes invalid
                                when corresponding window is closed.

The height attribute is writable only if the screen is split horizontally.
The width attribute is writable only if the screen is split vertically.

Window object type is available using "Window" attribute of vim module.

==============================================================================
6. Tab page objects                                         *python-tabpage*

Tab page objects represent vim tab pages. You can obtain them in a number of
ways:
        - via vim.current.tabpage (|python-current|)
        - from indexing vim.tabpages (|python-tabpages|)

You can use this object to access tab page windows. They have no methods and
no sequence or other interfaces.

Tab page attributes are:
        number          The tab page number like the one returned by
                        |tabpagenr()|.
        windows         Like |python-windows|, but for current tab page.
        vars            The tab page |t:| variables.
        window          Current tabpage window.
        valid           True or False. Tab page object becomes invalid when
                        corresponding tab page is closed.

TabPage object type is available using "TabPage" attribute of vim module.

==============================================================================
7. vim.bindeval objects                         *python-bindeval-objects*

vim.Dictionary object                           *python-Dictionary*
    Dictionary-like object providing access to vim |Dictionary| type.
    Attributes:
        Attribute  Description ~
        locked     One of                       *python-.locked*
                    Value           Description ~
                    zero            Variable is not locked
                    vim.VAR_LOCKED  Variable is locked, but can be unlocked
                    vim.VAR_FIXED   Variable is locked and can't be unlocked
                   Read-write. You can unlock locked variable by assigning
                   `True` or `False` to this attribute. No recursive locking
                   is supported.
        scope      One of
                    Value              Description ~
                    zero               Dictionary is not a scope one
                    vim.VAR_DEF_SCOPE  |g:| or |l:| dictionary
```

```
                    vim.VAR_SCOPE      Other scope dictionary,
                                       see |internal-variables|
    Methods (note: methods do not support keyword arguments):
        Method      Description ~
        keys()      Returns a list with dictionary keys.
        values()    Returns a list with dictionary values.
        items()     Returns a list of 2-tuples with dictionary contents.
        update(iterable), update(dictionary), update(**kwargs)
                    Adds keys to dictionary.
        get(key[, default=None])
                    Obtain key from dictionary, returning the default if it is
                    not present.
        pop(key[, default])
                    Remove specified key from dictionary and return
                    corresponding value. If key is not found and default is
                    given returns the default, otherwise raises KeyError.
        popitem()
                    Remove random key from dictionary and return (key, value)
                    pair.
        has_key(key)
                    Check whether dictionary contains specified key, similar
                    to `key in dict`.

        __new__(), __new__(iterable), __new__(dictionary), __new__(update)
                    You can use `vim.Dictionary()` to create new vim
                    dictionaries. `d=vim.Dictionary(arg)` is the same as
                    `d=vim.bindeval('{}');d.update(arg)`. Without arguments
                    constructs empty dictionary.

    Examples: >
        d = vim.Dictionary(food="bar")          # Constructor
        d['a'] = 'b'                            # Item assignment
        print d['a']                            # getting item
        d.update({'c': 'd'})                    # .update(dictionary)
        d.update(e='f')                         # .update(**kwargs)
        d.update((('g', 'h'), ('i', 'j')))      # .update(iterable)
        for key in d.keys():                    # .keys()
        for val in d.values():                  # .values()
        for key, val in d.items():              # .items()
        print isinstance(d, vim.Dictionary)     # True
        for key in d:                           # Iteration over keys
        class Dict(vim.Dictionary):             # Subclassing
<
    Note: when iterating over keys you should not modify dictionary.

vim.List object                                 *python-List*
    Sequence-like object providing access to vim |List| type.
    Supports `.locked` attribute, see |python-.locked|. Also supports the
    following methods:
        Method          Description ~
        extend(item)    Add items to the list.

        __new__(), __new__(iterable)
                        You can use `vim.List()` to create new vim lists.
                        `l=vim.List(iterable)` is the same as
                        `l=vim.bindeval('[]');l.extend(iterable)`. Without
                        arguments constructs empty list.
    Examples: >
        l = vim.List("abc")             # Constructor, result: ['a', 'b', 'c']
        l.extend(['abc', 'def'])        # .extend() method
        print l[1:]                     # slicing
        l[:0] = ['ghi', 'jkl']          # slice assignment
```

```
        print l[0]                      # getting item
        l[0] = 'mno'                    # assignment
        for i in l:                     # iteration
        print isinstance(l, vim.List)   # True
        class List(vim.List):           # Subclassing

vim.Function object                              *python-Function*
    Function-like object, acting like vim |Funcref| object. Accepts special
    keyword argument `self`, see |Dictionary-function|. You can also use
    `vim.Function(name)` constructor, it is the same as
    `vim.bindeval('function(%s)'%json.dumps(name))`.

    Attributes (read-only):
        Attribute    Description ~
        name         Function name.
        args         `None` or a |python-List| object with arguments.  Note
                     that this is a copy of the arguments list, constructed
                     each time you request this attribute. Modifications made
                     to the list will be ignored (but not to the containers
                     inside argument list: this is like |copy()| and not
                     |deepcopy()|).
        self         `None` or a |python-Dictionary| object with self
                     dictionary. Note that explicit `self` keyword used when
                     calling resulting object overrides this attribute.
        auto_rebind  Boolean. True if partial created from this Python object
                     and stored in the Vim script dictionary should be
                     automatically rebound to the dictionary it is stored in
                     when this dictionary is indexed. Exposes Vim internal
                     difference between `dict.func` (auto_rebind=True) and
                     `function(dict.func,dict)` (auto_rebind=False). This
                     attribute makes no sense if `self` attribute is `None`.

    Constructor additionally accepts `args`, `self` and `auto_rebind`
    keywords.  If `args` and/or `self` argument is given then it constructs
    a partial, see |function()|.  `auto_rebind` is only used when `self`
    argument is given, otherwise it is assumed to be `True` regardless of
    whether it was given or not.  If `self` is given then it defaults to
    `False`.

    Examples: >
        f = vim.Function('tr')                  # Constructor
        print f('abc', 'a', 'b')                # Calls tr('abc', 'a', 'b')
        vim.command('''
            function DictFun() dict
                return self
            endfunction
        ''')
        f = vim.bindeval('function("DictFun")')
        print f(self={})                        # Like call('DictFun', [], {})
        print isinstance(f, vim.Function)       # True

        p = vim.Function('DictFun', self={})
        print f()
        p = vim.Function('tr', args=['abc', 'a'])
        print f('b')

==============================================================================
8. pyeval() and py3eval() Vim functions                  *python-pyeval*

To facilitate bi-directional interface, you can use |pyeval()| and |py3eval()|
functions to evaluate Python expressions and pass their values to Vim script.
|pyxeval()| is also available.
```

==
9. Dynamic loading *python-dynamic*

On MS-Windows and Unix the Python library can be loaded dynamically. The
|:version| output then includes |+python/dyn| or |+python3/dyn|.

This means that Vim will search for the Python DLL or shared library file only
when needed. When you don't use the Python interface you don't need it, thus
you can use Vim without this file.

MS-Windows ~

To use the Python interface the Python DLL must be in your search path. In a
console window type "path" to see what directories are used. The 'pythondll'
or 'pythonthreedll' option can be also used to specify the Python DLL.

The name of the DLL should match the Python version Vim was compiled with.
Currently the name for Python 2 is "python27.dll", that is for Python 2.7.
That is the default value for 'pythondll'. For Python 3 it is python35.dll
(Python 3.5). To know for sure edit "gvim.exe" and search for
"python\d*.dll\c".

Unix ~

The 'pythondll' or 'pythonthreedll' option can be used to specify the Python
shared library file instead of DYNAMIC_PYTHON_DLL or DYNAMIC_PYTHON3_DLL file
what were specified at compile time. The version of the shared library must
match the Python 2.x or Python 3 version Vim was compiled with.

==
10. Python 3 *python3*

:py3 *:python3*
The `:py3` and `:python3` commands work similar to `:python`. A simple check
if the `:py3` command is working: >
:py3 print("Hello")

To see what version of Python you have: >
:py3 import sys
:py3 print(sys.version)
< *:py3file*
The `:py3file` command works similar to `:pyfile`.
:py3do
The `:py3do` command works similar to `:pydo`.

Vim can be built in four ways (:version output):
1. No Python support (-python, -python3)
2. Python 2 support only (+python or +python/dyn, -python3)
3. Python 3 support only (-python, +python3 or +python3/dyn)
4. Python 2 and 3 support (+python/dyn, +python3/dyn)

Some more details on the special case 4: *python-2-and-3*

When Python 2 and Python 3 are both supported they must be loaded dynamically.

When doing this on Linux/Unix systems and importing global symbols, this leads
to a crash when the second Python version is used. So either global symbols
are loaded but only one Python version is activated, or no global symbols are

```
loaded. The latter makes Python's "import" fail on libraries that expect the
symbols to be provided by Vim.
                                                        *E836* *E837*
Vim's configuration script makes a guess for all libraries based on one
standard Python library (termios).  If importing this library succeeds for
both Python versions, then both will be made available in Vim at the same
time.  If not, only the version first used in a session will be enabled.
When trying to use the other one you will get the E836 or E837 error message.

Here Vim's behavior depends on the system in which it was configured.  In a
system where both versions of Python were configured with --enable-shared,
both versions of Python will be activated at the same time.  There will still
be problems with other third party libraries that were not linked to
libPython.

To work around such problems there are these options:
1. The problematic library is recompiled to link to the according
   libpython.so.
2. Vim is recompiled for only one Python version.
3. You undefine PY_NO_RTLD_GLOBAL in auto/config.h after configuration.  This
   may crash Vim though.

                                                        *E880*
Raising SystemExit exception in python isn't endorsed way to quit vim, use: >
        :py vim.command("qall!")
<

                                                        *has-python*
You can test what Python version is available with: >
        if has('python')
          echo 'there is Python 2.x'
        elseif has('python3')
          echo 'there is Python 3.x'
        endif

Note however, that when Python 2 and 3 are both available and loaded
dynamically, these has() calls will try to load them.  If only one can be
loaded at a time, just checking if Python 2 or 3 are available will prevent
the other one from being available.

==============================================================================
11. Python X                                            *python_x* *pythonx*

Because most python code can be written so that it works with python 2.6+ and
python 3 the pyx* functions and commands have been written.  They work exactly
the same as the Python 2 and 3 variants, but select the Python version using
the 'pyxversion' setting.

You should set 'pyxversion' in your |.vimrc| to prefer Python 2 or Python 3
for Python commands. If you change this setting at runtime you may risk that
state of plugins (such as initialization) may be lost.

If you want to use a module, you can put it in the {rtp}/pythonx directory.
See |pythonx-directory|.

                                                        *:pyx* *:pythonx*
The `:pyx` and `:pythonx` commands work similar to `:python`.  A simple check
if the `:pyx` command is working: >
        :pyx print("Hello")

To see what version of Python is being used: >
        :pyx import sys
```

```
        :pyx print(sys.version)
<
                                          *:pyxfile* *python_x-special-comments*
The `:pyxfile` command works similar to `:pyfile`.  However you can add one of
these comments to force Vim using `:pyfile` or `:py3file`: >
  #!/any string/python2         " Shebang. Must be the first line of the file.
  #!/any string/python3         " Shebang. Must be the first line of the file.
  # requires python 2.x         " Maximum lines depend on 'modelines'.
  # requires python 3.x         " Maximum lines depend on 'modelines'.
Unlike normal modelines, the bottom of the file is not checked.
If none of them are found, the 'pyxversion' setting is used.
                                                        *W20* *W21*
If Vim does not support the selected Python version a silent message will be
printed.  Use `:messages` to read them.

                                                        *:pyxdo*
The `:pyxdo` command works similar to `:pydo`.

                                                        *has-pythonx*
You can test if pyx* commands are available with: >
        if has('pythonx')
          echo 'pyx* commands are available. (Python ' . &pyx . ')'
        endif

When compiled with only one of |+python| or |+python3|, the has() returns 1.
When compiled with both |+python| and |+python3|, the test depends on the
'pyxversion' setting.  If 'pyxversion' is 0, it tests Python 3 first, and if
it is not available then Python 2.  If 'pyxversion' is 2 or 3, it tests only
Python 2 or 3 respectively.

Note that for `has('pythonx')` to work it may try to dynamically load Python 3
or 2.  This may have side effects, especially when Vim can only load one of
the two.

If a user prefers Python 2 and want to fallback to Python 3, he needs to set
'pyxversion' explicitly in his |.vimrc|.  E.g.: >
        if has('python')
          set pyx=2
        elseif has('python3')
          set pyx=3
        endif

==============================================================================
12. Building with Python support                        *python-building*

A few hints for building with Python 2 or 3 support.

UNIX

See src/Makefile for how to enable including the Python interface.

On Ubuntu you will want to install these packages for Python 2:
        python
        python-dev
For Python 3:
        python3
        python3-dev
For Python 3.6:
        python3.6
        python3.6-dev

If you have more than one version of Python 3, you need to link python3 to the
```

```
one you prefer, before running configure.

==============================================================================
 vim:tw=78:ts=8:ft=help:norl:
*if_tcl.txt*    For Vim version 8.0.  Last change: 2016 Jan 01


                  VIM REFERENCE MANUAL    by Ingo Wilken


The Tcl Interface to Vim                                *tcl* *Tcl* *TCL*

1. Commands                             |tcl-ex-commands|
2. Tcl commands                         |tcl-commands|
3. Tcl variables                        |tcl-variables|
4. Tcl window commands                  |tcl-window-cmds|
5. Tcl buffer commands                  |tcl-buffer-cmds|
6. Miscellaneous; Output from Tcl       |tcl-misc| |tcl-output|
7. Known bugs & problems                |tcl-bugs|
8. Examples                             |tcl-examples|
9. Dynamic loading                      |tcl-dynamic|

{Vi does not have any of these commands} *E280*

The Tcl interface only works when Vim was compiled with the |+tcl| feature.

WARNING: There are probably still some bugs.  Please send bug reports,
comments, ideas etc to <Ingo.Wilken@informatik.uni-oldenburg.de>

==============================================================================
1. Commands                             *tcl-ex-commands* *E571* *E572*

                                                        *:tcl* *:tc*
:tc[l] {cmd}            Execute Tcl command {cmd}.  A simple check if `:tcl`
                        is working: >
                                :tcl puts "Hello"

:[range]tc[l] << {endmarker}
{script}
{endmarker}
                        Execute Tcl script {script}.
                        Note: This command doesn't work when the Tcl feature
                        wasn't compiled in.  To avoid errors, see
                        |script-here|.

{endmarker} must NOT be preceded by any white space.  If {endmarker} is
omitted from after the "<<", a dot '.' must be used after {script}, like for
the |:append| and |:insert| commands.
This form of the |:tcl| command is mainly useful for including tcl code in Vim
scripts.

Example: >
        function! DefineDate()
            tcl << EOF
            proc date {} {
                return [clock format [clock seconds]]
            }
        EOF
        endfunction
<
To see what version of Tcl you have: >
        :tcl puts [info patchlevel]
```

<

                                                    :tcldo *:tcld*
:[range]tcld[o] {cmd}   Execute Tcl command {cmd} for each line in [range]
                        with the variable "line" being set to the text of each
                        line in turn, and "lnum" to the line number. Setting
                        "line" will change the text, but note that it is not
                        possible to add or delete lines using this command.
                        If {cmd} returns an error, the command is interrupted.
                        The default for [range] is the whole file: "1,$".
                        See |tcl-var-line| and |tcl-var-lnum|. {not in Vi}

                                                    :tclfile *:tclf*
:tclf[ile] {file}   Execute the Tcl script in {file}. This is the same as
                        ":tcl source {file}", but allows file name completion.
                        {not in Vi}

Note that Tcl objects (like variables) persist from one command to the next,
just as in the Tcl shell.

Executing Tcl commands is not possible in the |sandbox|.

==
2. Tcl commands   *tcl-commands*

Tcl code gets all of its access to vim via commands in the "::vim" namespace.
The following commands are implemented: >

        ::vim::beep   # Guess.
        ::vim::buffer {n}   # Create Tcl command for one buffer.
        ::vim::buffer list   # Create Tcl commands for all buffers.
        ::vim::command [-quiet] {cmd}   # Execute an Ex command.
        ::vim::expr {expr}   # Use Vim's expression evaluator.
        ::vim::option {opt}   # Get vim option.
        ::vim::option {opt} {val}   # Set vim option.
        ::vim::window list   # Create Tcl commands for all windows.

Commands:
        ::vim::beep   *tcl-beep*
        Honk. Does not return a result.

        ::vim::buffer {n}   *tcl-buffer*
        ::vim::buffer exists {n}
        ::vim::buffer list
        Provides access to vim buffers. With an integer argument, creates a
        buffer command (see |tcl-buffer-cmds|) for the buffer with that
        number, and returns its name as the result. Invalid buffer numbers
        result in a standard Tcl error. To test for valid buffer numbers,
        vim's internal functions can be used: >
                set nbufs [::vim::expr bufnr("$")]
                set isvalid [::vim::expr "bufexists($n)"]
<   The "list" option creates a buffer command for each valid buffer, and
        returns a list of the command names as the result.
        Example: >
                set bufs [::vim::buffer list]
                foreach b $bufs { $b append end "The End!" }
<   The "exists" option checks if a buffer with the given number exists.
        Example: >
                if { [::vim::buffer exists $n] } { ::vim::command ":e #$n" }
<   This command might be replaced by a variable in future versions.
        See also |tcl-var-current| for the current buffer.

```
        ::vim::command {cmd}                            *tcl-command*
        ::vim::command -quiet {cmd}
        Execute the vim (ex-mode) command {cmd}.  Any Ex command that affects
        a buffer or window uses the current buffer/current window.  Does not
        return a result other than a standard Tcl error code.  After this
        command is completed, the "::vim::current" variable is updated.
        The "-quiet" flag suppresses any error messages from vim.
        Examples: >
                ::vim::command "set ts=8"
                ::vim::command "%s/foo/bar/g"
<       To execute normal-mode commands, use "normal" (see |:normal|): >
                set cmd "jj"
                ::vim::command "normal $cmd"
<       See also |tcl-window-command| and |tcl-buffer-command|.

        ::vim::expr {expr}                              *tcl-expr*
        Evaluates the expression {expr} using vim's internal expression
        evaluator (see |expression|).   Any expression that queries a buffer
        or window property uses the current buffer/current window.  Returns
        the result as a string.  A |List| is turned into a string by joining
        the items and inserting line breaks.
        Examples: >
                set perl_available [::vim::expr has("perl")]
<       See also |tcl-window-expr| and |tcl-buffer-expr|.

        ::vim::option {opt}                             *tcl-option*
        ::vim::option {opt} {value}
        Without second argument, queries the value of a vim option.  With this
        argument, sets the vim option to {value}, and returns the previous
        value as the result.  Any options that are marked as 'local to buffer'
        or 'local to window' affect the current buffer/current window.  The
        global value is not changed, use the ":set" command for that.  For
        boolean options, {value} should be "0" or "1", or any of the keywords
        "on", "off" or "toggle".  See |option-summary| for a list of options.
        Example: >
                ::vim::option ts 8
<       See also |tcl-window-option| and |tcl-buffer-option|.

        ::vim::window {option}                          *tcl-window*
        Provides access to vim windows.  Currently only the "list" option is
        implemented.  This creates a window command (see |tcl-window-cmds|) for
        each window, and returns a list of the command names as the result.
        Example: >
                set wins [::vim::window list]
                foreach w $wins { $w height 4 }
<       This command might be replaced by a variable in future versions.
        See also |tcl-var-current| for the current window.

==============================================================================
3. Tcl variables                                        *tcl-variables*

The ::vim namespace contains a few variables.  These are created when the Tcl
interpreter is called from vim and set to current values. >

        ::vim::current          # array containing "current" objects
        ::vim::lbase            # number of first line
        ::vim::range            # array containing current range numbers
        line                    # current line as a string (:tcldo only)
        lnum                    # current line number (:tcldo only)

Variables:
```

```
        ::vim::current                                  *tcl-var-current*
        This is an array providing access to various "current" objects
        available in vim.  The contents of this array are updated after
        "::vim::command" is called, as this might change vim's current
        settings (e.g., by deleting the current buffer).
        The "buffer" element contains the name of the buffer command for the
        current buffer.  This can be used directly to invoke buffer commands
        (see |tcl-buffer-cmds|).  This element is read-only.
        Example: >
                $::vim::current(buffer) insert begin "Hello world"
<       The "window" element contains the name of the window command for the
        current window.  This can be used directly to invoke window commands
        (see |tcl-window-cmds|).  This element is read-only.
        Example: >
                $::vim::current(window) height 10
<
        ::vim::lbase                                    *tcl-var-lbase*
        This variable controls how Tcl treats line numbers.  If it is set to
        '1', then lines and columns start at 1.  This way, line numbers from
        Tcl commands and vim expressions are compatible.  If this variable is
        set to '0', then line numbers and columns start at 0 in Tcl.  This is
        useful if you want to treat a buffer as a Tcl list or a line as a Tcl
        string and use standard Tcl commands that return an index ("lsort" or
        "string first", for example).  The default value is '1'.  Currently,
        any non-zero values is treated as '1', but your scripts should not
        rely on this.  See also |tcl-linenumbers|.

        ::vim::range                                    *tcl-var-range*
        This is an array with three elements, "start", "begin" and "end".  It
        contains the line numbers of the start and end row of the current
        range.  "begin" is the same as "start".  This variable is read-only.
        See |tcl-examples|.

        line                                            *tcl-var-line*
        lnum                                            *tcl-var-lnum*
        These global variables are only available if the ":tcldo" Ex command
        is being executed.  They contain the text and line number of the
        current line.  When the Tcl command invoked by ":tcldo" is completed,
        the current line is set to the contents of the "line" variable, unless
        the variable was unset by the Tcl command.  The "lnum" variable is
        read-only.  These variables are not in the "::vim" namespace so they
        can be used in ":tcldo" without much typing (this might be changed in
        future versions).  See also |tcl-linenumbers|.

==============================================================================
4. Tcl window commands                                  *tcl-window-cmds*

Window commands represent vim windows.  They are created by several commands:
        ::vim::window list                      |tcl-window|
        "windows" option of a buffer command    |tcl-buffer-windows|
The ::vim::current(window) variable contains the name of the window command
for the current window.  A window command is automatically deleted when the
corresponding vim window is closed.

Let's assume the name of the window command is stored in the Tcl variable "win",
i.e. "$win" calls the command.  The following options are available: >

        $win buffer             # Create Tcl command for window's buffer.
        $win command {cmd}      # Execute Ex command in windows context.
        $win cursor             # Get current cursor position.
        $win cursor {var}       # Set cursor position from array variable.
        $win cursor {row} {col} # Set cursor position.
```

```
        $win delcmd {cmd}       # Call Tcl command when window is closed.
        $win expr {expr}        # Evaluate vim expression in windows context.
        $win height             # Report the window's height.
        $win height {n}         # Set the window's height.
        $win option {opt} [val] # Get/Set vim option in windows context.

Options:
        $win buffer                                         *tcl-window-buffer*
        Creates a Tcl command for the window's buffer, and returns its name as
        the result.  The name should be stored in a variable: >
                set buf [$win buffer]
<       $buf is now a valid Tcl command.  See |tcl-buffer-cmds| for the
        available options.

        $win cursor                                         *tcl-window-cursor*
        $win cursor {var}
        $win cursor {row} {col}
        Without argument, reports the current cursor position as a string.
        This can be converted to a Tcl array variable: >
                array set here [$win cursor]
<       "here(row)" and "here(column)" now contain the cursor position.
        With a single argument, the argument is interpreted as the name of a
        Tcl array variable, which must contain two elements "row" and "column".
        These are used to set the cursor to the new position: >
                $win cursor here        ;# not $here !
<       With two arguments, sets the cursor to the specified row and column: >
                $win cursor $here(row) $here(column)
<       Invalid positions result in a standard Tcl error, which can be caught
        with "catch".  The row and column values depend on the "::vim::lbase"
        variable.  See |tcl-var-lbase|.

        $win delcmd {cmd}                                   *tcl-window-delcmd*
        Registers the Tcl command {cmd} as a deletion callback for the window.
        This command is executed (in the global scope) just before the window
        is closed.  Complex commands should be build with "list": >
                $win delcmd [list puts vimerr "window deleted"]
<       See also |tcl-buffer-delcmd|.

        $win height                                         *tcl-window-height*
        $win height {n}
        Without argument, reports the window's current height.  With an
        argument, tries to set the window's height to {n}, then reports the
        new height (which might be different from {n}).

        $win command [-quiet] {cmd}                         *tcl-window-command*
        $win expr {expr}                                    *tcl-window-expr*
        $win option {opt} [val]                             *tcl-window-option*
        These are similar to "::vim::command" etc., except that everything is
        done in the context of the window represented by $win, instead of the
        current window.  For example, setting an option that is marked 'local
        to window' affects the window $win.  Anything that affects or queries
        a buffer uses the buffer displayed in this window (i.e. the buffer
        that is represented by "$win buffer").  See |tcl-command|, |tcl-expr|
        and |tcl-option| for more information.
        Example: >
                $win option number on

==============================================================================
5. Tcl buffer commands                                  *tcl-buffer-cmds*

Buffer commands represent vim buffers.  They are created by several commands:
        ::vim::buffer {N}                       |tcl-buffer|
```

```
        ::vim::buffer list                      |tcl-buffer|
        "buffer" option of a window command     |tcl-window-buffer|
The ::vim::current(buffer) variable contains the name of the buffer command
for the current buffer.  A buffer command is automatically deleted when the
corresponding vim buffer is destroyed.  Whenever the buffer's contents are
changed, all marks in the buffer are automatically adjusted.  Any changes to
the buffer's contents made by Tcl commands can be undone with the "undo" vim
command (see |undo|).

Let's assume the name of the buffer command is stored in the Tcl variable "buf",
i.e. "$buf" calls the command.  The following options are available: >

        $buf append {n} {str}   # Append a line to buffer, after line {n}.
        $buf command {cmd}      # Execute Ex command in buffers context.
        $buf count              # Report number of lines in buffer.
        $buf delcmd {cmd}       # Call Tcl command when buffer is deleted.
        $buf delete {n}         # Delete a single line.
        $buf delete {n} {m}     # Delete several lines.
        $buf expr {expr}        # Evaluate vim expression in buffers context.
        $buf get {n}            # Get a single line as a string.
        $buf get {n} {m}        # Get several lines as a list.
        $buf insert {n} {str}   # Insert a line in buffer, as line {n}.
        $buf last               # Report line number of last line in buffer.
        $buf mark {mark}        # Report position of buffer mark.
        $buf name               # Report name of file in buffer.
        $buf number             # Report number of this buffer.
        $buf option {opt} [val] # Get/Set vim option in buffers context.
        $buf set {n} {text}     # Replace a single line.
        $buf set {n} {m} {list} # Replace several lines.
        $buf windows            # Create Tcl commands for buffer's windows.
<
                                                        *tcl-linenumbers*
Most buffer commands take line numbers as arguments.  How Tcl treats these
numbers depends on the "::vim::lbase" variable (see |tcl-var-lbase|).  Instead
of line numbers, several keywords can be also used: "top", "start", "begin",
"first", "bottom", "end" and "last".

Options:
        $buf append {n} {str}                           *tcl-buffer-append*
        $buf insert {n} {str}                           *tcl-buffer-insert*
        Add a line to the buffer.  With the "insert" option, the string
        becomes the new line {n}, with "append" it is inserted after line {n}.
        Example: >
                $buf insert top "This is the beginning."
                $buf append end "This is the end."
<       To add a list of lines to the buffer, use a loop: >
                foreach line $list { $buf append $num $line ; incr num }
<
        $buf count                                      *tcl-buffer-count*
        Reports the total number of lines in the buffer.

        $buf delcmd {cmd}                               *tcl-buffer-delcmd*
        Registers the Tcl command {cmd} as a deletion callback for the buffer.
        This command is executed (in the global scope) just before the buffer
        is deleted.  Complex commands should be build with "list": >
                $buf delcmd [list puts vimerr "buffer [$buf number] gone"]
<       See also |tcl-window-delcmd|.

        $buf delete {n}                                 *tcl-buffer-delete*
        $buf delete {n} {m}
        Deletes line {n} or lines {n} through {m} from the buffer.
        This example deletes everything except the last line: >
```

```
                $buf delete first [expr [$buf last] - 1]
<
        $buf get {n}                                        *tcl-buffer-get*
        $buf get {n} {m}
        Gets one or more lines from the buffer.  For a single line, the result
        is a string; for several lines, a list of strings.
        Example: >
                set topline [$buf get top]
<
        $buf last                                           *tcl-buffer-last*
        Reports the line number of the last line.  This value depends on the
        "::vim::lbase" variable.  See |tcl-var-lbase|.

        $buf mark {mark}                                    *tcl-buffer-mark*
        Reports the position of the named mark as a string, similar to the
        cursor position of the "cursor" option of a window command (see
        |tcl-window-cursor|).  This can be converted to a Tcl array variable: >
                array set mpos [$buf mark "a"]
<       "mpos(column)" and "mpos(row)" now contain the position of the mark.
        If the mark is not set, a standard Tcl error results.

        $buf name
        Reports the name of the file in the buffer.  For a buffer without a
        file, this is an empty string.

        $buf number
        Reports the number of this buffer.  See |:buffers|.
        This example deletes a buffer from vim: >
                ::vim::command "bdelete [$buf number]"
<
        $buf set {n} {string}                               *tcl-buffer-set*
        $buf set {n} {m} {list}
        Replace one or several lines in the buffer.  If the list contains more
        elements than there are lines to replace, they are inserted into the
        buffer.  If the list contains fewer elements, any unreplaced line is
        deleted from the buffer.

        $buf windows                                        *tcl-buffer-windows*
        Creates a window command for each window that displays this buffer, and
        returns a list of the command names as the result.
        Example: >
                set winlist [$buf windows]
                foreach win $winlist { $win height 4 }
<       See |tcl-window-cmds| for the available options.

        $buf command [-quiet] {cmd}                         *tcl-buffer-command*
        $buf expr {expr}                                    *tcl-buffer-expr*
        $buf option {opt} [val]                             *tcl-buffer-option*
        These are similar to "::vim::command" etc., except that everything is
        done in the context of the buffer represented by $buf, instead of the
        current buffer.  For example, setting an option that is marked 'local
        to buffer' affects the buffer $buf.  Anything that affects or queries
        a window uses the first window in vim's window list that displays this
        buffer (i.e. the first entry in the list returned by "$buf windows").
        See |tcl-command|, |tcl-expr| and |tcl-option| for more information.
        Example: >
                if { [$buf option modified] } { $buf command "w" }

==============================================================================
6. Miscellaneous; Output from Tcl               *tcl-misc* *tcl-output*

The standard Tcl commands "exit" and "catch" are replaced by custom versions.
```

```
"exit" terminates the current Tcl script and returns to vim, which deletes the
Tcl interpreter.  Another call to ":tcl" then creates a new Tcl interpreter.
"exit" does NOT terminate vim!  "catch" works as before, except that it does
not prevent script termination from "exit".  An exit code != 0 causes the ex
command that invoked the Tcl script to return an error.

Two new I/O streams are available in Tcl, "vimout" and "vimerr".  All output
directed to them is displayed in the vim message area, as information messages
and error messages, respectively.  The standard Tcl output streams stdout and
stderr are mapped to vimout and vimerr, so that a normal "puts" command can be
used to display messages in vim.

==============================================================================
7. Known bugs & problems                                    *tcl-bugs*

Calling one of the Tcl Ex commands from inside Tcl (via "::vim::command") may
have unexpected side effects.  The command creates a new interpreter, which
has the same abilities as the standard interpreter - making "::vim::command"
available in a safe child interpreter therefore makes the child unsafe.  (It
would be trivial to block nested :tcl* calls or ensure that such calls from a
safe interpreter create only new safe interpreters, but quite pointless -
depending on vim's configuration, "::vim::command" may execute arbitrary code
in any number of other scripting languages.)  A call to "exit" within this new
interpreter does not affect the old interpreter; it only terminates the new
interpreter, then script processing continues normally in the old interpreter.

Input from stdin is currently not supported.

==============================================================================
8. Examples:                                                *tcl-examples*

Here are a few small (and maybe useful) Tcl scripts.

This script sorts the lines of the entire buffer (assume it contains a list
of names or something similar):
        set buf $::vim::current(buffer)
        set lines [$buf get top bottom]
        set lines [lsort -dictionary $lines]
        $buf set top bottom $lines

This script reverses the lines in the buffer.  Note the use of "::vim::lbase"
and "$buf last" to work with any line number setting.
        set buf $::vim::current(buffer)
        set t $::vim::lbase
        set b [$buf last]
        while { $t < $b } {
                set tl [$buf get $t]
                set bl [$buf get $b]
                $buf set $t $bl
                $buf set $b $tl
                incr t
                incr b -1
        }

This script adds a consecutive number to each line in the current range:
        set buf $::vim::current(buffer)
        set i $::vim::range(start)
        set n 1
        while { $i <= $::vim::range(end) } {
                set line [$buf get $i]
                $buf set $i "$n\t$line"
                incr i ; incr n
```

```
	}

The same can also be done quickly with two Ex commands, using ":tcldo":
	:tcl set n 1
	:[range]tcldo set line "$n\t$line" ; incr n

This procedure runs an Ex command on each buffer (idea stolen from Ron Aaron):
	proc eachbuf { cmd } {
		foreach b [::vim::buffer list] {
			$b command $cmd
		}
	}
Use it like this:
	:tcl eachbuf %s/foo/bar/g
Be careful with Tcl's string and backslash substitution, tough.  If in doubt,
surround the Ex command with curly braces.


If you want to add some Tcl procedures permanently to vim, just place them in
a file (e.g. "~/.vimrc.tcl" on Unix machines), and add these lines to your
startup file (usually "~/.vimrc" on Unix):
	if has("tcl")
		tclfile ~/.vimrc.tcl
	endif

==============================================================================
9. Dynamic loading                                        *tcl-dynamic*

On MS-Windows and Unix the Tcl library can be loaded dynamically.  The
|:version| output then includes |+tcl/dyn|.

This means that Vim will search for the Tcl DLL or shared library file only
when needed.  When you don't use the Tcl interface you don't need it, thus you
can use Vim without this file.


MS-Windows ~

To use the Tcl interface the Tcl DLL must be in your search path.  In a
console window type "path" to see what directories are used.  The 'tcldll'
option can be also used to specify the Tcl DLL.

The name of the DLL must match the Tcl version Vim was compiled with.
Currently the name is "tcl86.dll".  That is for Tcl 8.6.  To know for sure
edit "gvim.exe" and search for "tcl\d*.dll\c".


Unix ~

The 'tcldll' option can be used to specify the Tcl shared library file instead
of DYNAMIC_TCL_DLL file what was specified at compile time.  The version of
the shared library must match the Tcl version Vim was compiled with.

==============================================================================
 vim:tw=78:ts=8:ft=help:norl:
*if_ole.txt*    For Vim version 8.0.  Last change: 2008 Aug 16


                  VIM REFERENCE MANUAL    by Paul Moore


The OLE Interface to Vim                                  *ole-interface*
```

{Vi does not have any of these commands}

OLE is only available when compiled with the |+ole| feature. See
src/if_ole.INSTALL.
An alternative is using the client-server communication |clientserver|.

==
1. Activation *ole-activation*

Vim acts as an OLE automation server, accessible from any automation client,
for example, Visual Basic, Python, or Perl. The Vim application "name" (its
"ProgID", in OLE terminology) is "Vim.Application".

Hence, in order to start a Vim instance (or connect to an already running
instance), code similar to the following should be used:

```
[Visual Basic] >
        Dim Vim As Object
        Set Vim = CreateObject("Vim.Application")

[Python] >
        from win32com.client.dynamic import Dispatch
        vim = Dispatch('Vim.Application')

[Perl] >
        use Win32::OLE;
        $vim = new Win32::OLE 'Vim.Application';

[C#] >
        // Add a reference to Vim in your project.
        // Choose the COM tab.
        // Select "Vim Ole Interface 1.1 Type Library"
        Vim.Vim vimobj = new Vim.Vim();
```

Vim does not support acting as a "hidden" OLE server, like some other OLE
Automation servers. When a client starts up an instance of Vim, that instance
is immediately visible. Simply closing the OLE connection to the Vim instance
is not enough to shut down the Vim instance - it is necessary to explicitly
execute a quit command (for example, :qa!, :wqa).

==
2. Methods *ole-methods*

Vim exposes four methods for use by clients.

ole-sendkeys
SendKeys(keys) Execute a series of keys.

This method takes a single parameter, which is a string of keystrokes. These
keystrokes are executed exactly as if they had been types in at the keyboard.
Special keys can be given using their <..> names, as for the right hand side
of a mapping. Note: Execution of the Ex "normal" command is not supported -
see below |ole-normal|.

Examples (Visual Basic syntax) >

```
        Vim.SendKeys "ihello<Esc>"
        Vim.SendKeys "ma1GV4jy`a"

These examples assume that Vim starts in Normal mode.  To force Normal mode,
start the key sequence with CTRL-\ CTRL-N as in >

        Vim.SendKeys "<C-\><C-N>ihello<Esc>"

CTRL-\ CTRL-N returns Vim to Normal mode, when in Insert or Command-line mode.
Note that this doesn't work halfway a Vim command

                                                        *ole-eval*
Eval(expr)              Evaluate an expression.

This method takes a single parameter, which is an expression in Vim's normal
format (see |expression|).  It returns a string, which is the result of
evaluating the expression.  A |List| is turned into a string by joining the
items and inserting line breaks.

Examples (Visual Basic syntax) >
        Line20 = Vim.Eval("getline(20)")
        Twelve = Vim.Eval("6 + 6")              ' Note this is a STRING
        Font = Vim.Eval("&guifont")
<
                                                        *ole-setforeground*
SetForeground()         Make the Vim window come to the foreground

This method takes no arguments.  No value is returned.

Example (Visual Basic syntax) >
        Vim.SetForeground
<

                                                        *ole-gethwnd*
GetHwnd()               Return the handle of the Vim window.

This method takes no arguments.  It returns the hwnd of the main Vimwindow.
You can use this if you are writing something which needs to manipulate the
Vim window, or to track it in the z-order, etc.

Example (Visual Basic syntax) >
        Vim_Hwnd = Vim.GetHwnd
<

==============================================================================
3. The "normal" command                                 *ole-normal*

Due to the way Vim processes OLE Automation commands, combined with the method
of implementation of the Ex command :normal, it is not possible to execute the
:normal command via OLE automation.  Any attempt to do so will fail, probably
harmlessly, although possibly in unpredictable ways.

There is currently no practical way to trap this situation, and users must
simply be aware of the limitation.
==============================================================================
4. Registration                                 *ole-registration* *E243*

Before Vim will act as an OLE server, it must be registered in the system
registry.  In order to do this, Vim should be run with a single parameter of
"-register".
                                                        *-register*  >
        gvim -register
```

If gvim with OLE support is run and notices that no Vim OLE server has been
registered, it will present a dialog and offers you the choice to register by
clicking "Yes".

In some situations registering is not possible. This happens when the
registry is not writable. If you run into this problem you need to run gvim
as "Administrator".

Once vim is registered, the application path is stored in the registry.
Before moving, deleting, or upgrading Vim, the registry entries should be
removed using the "-unregister" switch.
-unregister >

```
gvim -unregister
```

The OLE mechanism will use the first registered Vim it finds. If a Vim is
already running, this one will be used. If you want to have (several) Vim
sessions open that should not react to OLE commands, use the non-OLE version,
and put it in a different directory. The OLE version should then be put in a
directory that is not in your normal path, so that typing "gvim" will start
the non-OLE version.

-silent

To avoid the message box that pops up to report the result, prepend "-silent":
>

```
gvim -silent -register
gvim -silent -unregister
```

==========

5. MS Visual Studio integration *MSVisualStudio* *VisVim*

The OLE version can be used to run Vim as the editor in Microsoft Visual
Studio. This is called "VisVim". It is included in the archive that contains
the OLE version. The documentation can be found in the runtime directory, the
README_VisVim.txt file.

Using Vim with Visual Studio .Net~

With .Net you no longer really need VisVim, since .Net studio has support for
external editors. Follow these directions:

In .Net Studio choose from the menu Tools->External Tools...
Add

```
     Title     - Vim
     Command   - c:\vim\vim63\gvim.exe
     Arguments - --servername VS_NET --remote-silent "+call cursor($(CurLine), $
(CurCol))" $(ItemPath)
     Init Dir  - Empty
```

Now, when you open a file in .Net, you can choose from the .Net menu:
Tools->Vim

That will open the file in Vim.
You can then add this external command as an icon and place it anywhere you
like. You might also be able to set this as your default editor.

If you refine this further, please post back to the Vim maillist so we have a
record of it.

--servername VS_NET
This will create a new instance of vim called VS_NET. So if you open multiple

```
files from VS, they will use the same instance of Vim.  This allows you to
have multiple copies of Vim running, but you can control which one has VS
files in it.

--remote-silent "+call cursor(10, 27)"
              - Places the cursor on line 10 column 27
In Vim >
   :h --remote-silent for more details

[.Net remarks provided by Dave Fishburn and Brian Sturk]

==============================================================================
 vim:tw=78:ts=8:ft=help:norl:
*if_ruby.txt*   For Vim version 8.0.  Last change: 2016 Sep 01


                  VIM REFERENCE MANUAL    by Shugo Maeda

The Ruby Interface to Vim                                   *ruby* *Ruby*


1. Commands                       |ruby-commands|
2. The Vim module                 |ruby-vim|
3. Vim::Buffer objects            |ruby-buffer|
4. Vim::Window objects            |ruby-window|
5. Global variables               |ruby-globals|
6. Dynamic loading                |ruby-dynamic|

{Vi does not have any of these commands}
                        *E266* *E267* *E268* *E269* *E270* *E271* *E272* *E273*

The Ruby interface only works when Vim was compiled with the |+ruby| feature.

The home page for ruby is http://www.ruby-lang.org/.  You can find links for
downloading Ruby there.

==============================================================================
1. Commands                                             *ruby-commands*

                                                        *:ruby* *:rub*
:rub[y] {cmd}           Execute Ruby command {cmd}.  A command to try it out: >
                                :ruby print "Hello"

:rub[y] << {endpattern}
{script}
{endpattern}
                        Execute Ruby script {script}.
                        {endpattern} must NOT be preceded by any white space.
                        If {endpattern} is omitted, it defaults to a dot '.'
                        like for the |:append| and |:insert| commands.  This
                        form of the |:ruby| command is mainly useful for
                        including ruby code in vim scripts.
                        Note: This command doesn't work when the Ruby feature
                        wasn't compiled in.  To avoid errors, see
                        |script-here|.

Example Vim script: >

        function! RedGem()
        ruby << EOF
        class Garnet
                def initialize(s)
```

```
                        @buffer = Vim::Buffer.current
                        vimputs(s)
                end
                def vimputs(s)
                        @buffer.append(@buffer.count,s)
                end
        end
        gem = Garnet.new("pretty")
        EOF
        endfunction
<
To see what version of Ruby you have: >
        :ruby print RUBY_VERSION
<

                                                *:rubydo* *:rubyd* *E265*
:[range]rubyd[o] {cmd}  Evaluate Ruby command {cmd} for each line in the
                        [range], with $_ being set to the text of each line in
                        turn, without a trailing <EOL>.  Setting $_ will change
                        the text, but note that it is not possible to add or
                        delete lines using this command.
                        The default for [range] is the whole file: "1,$".

                                                        *:rubyfile* *:rubyf*
:rubyf[ile] {file}      Execute the Ruby script in {file}.  This is the same as
                        `:ruby load 'file'`, but allows file name completion.

Executing Ruby commands is not possible in the |sandbox|.

==============================================================================
2. The Vim module                                       *ruby-vim*

Ruby code gets all of its access to vim via the "Vim" module.

Overview: >
        print "Hello"                          # displays a message
        Vim.command(cmd)                      # execute an Ex command
        num = Vim::Window.count               # gets the number of windows
        w = Vim::Window[n]                    # gets window "n"
        cw = Vim::Window.current              # gets the current window
        num = Vim::Buffer.count               # gets the number of buffers
        b = Vim::Buffer[n]                    # gets buffer "n"
        cb = Vim::Buffer.current              # gets the current buffer
        w.height = lines                      # sets the window height
        w.cursor = [row, col]                 # sets the window cursor position
        pos = w.cursor                        # gets an array [row, col]
        name = b.name                         # gets the buffer file name
        line = b[n]                           # gets a line from the buffer
        num = b.count                         # gets the number of lines
        b[n] = str                            # sets a line in the buffer
        b.delete(n)                           # deletes a line
        b.append(n, str)                      # appends a line after n
        line = Vim::Buffer.current.line       # gets the current line
        num = Vim::Buffer.current.line_number # gets the current line number
        Vim::Buffer.current.line = "test"     # sets the current line number
<

Module Functions:

                                                        *ruby-message*
Vim::message({msg})
        Displays the message {msg}.
```

ruby-set_option
Vim::set_option({arg})
Sets a vim option. {arg} can be any argument that the ":set" command accepts. Note that this means that no spaces are allowed in the argument! See |:set|.

ruby-command
Vim::command({cmd})
Executes Ex command {cmd}.

ruby-evaluate
Vim::evaluate({expr})
Evaluates {expr} using the vim internal expression evaluator (see |expression|). Returns the expression result as:
- a Integer if the Vim expression evaluates to a number
- a Float if the Vim expression evaluates to a float
- a String if the Vim expression evaluates to a string
- a Array if the Vim expression evaluates to a Vim list
- a Hash if the Vim expression evaluates to a Vim dictionary
Dictionaries and lists are recursively expanded.

==
3. Vim::Buffer objects *ruby-buffer*

Vim::Buffer objects represent vim buffers.

Class Methods:

current Returns the current buffer object.
count Returns the number of buffers.
self[{n}] Returns the buffer object for the number {n}. The first number is 0.

Methods:

name Returns the name of the buffer.
number Returns the number of the buffer.
count Returns the number of lines.
length Returns the number of lines.
self[{n}] Returns a line from the buffer. {n} is the line number.
self[{n}] = {str}
Sets a line in the buffer. {n} is the line number.
delete({n}) Deletes a line from the buffer. {n} is the line number.
append({n}, {str})
Appends a line after the line {n}.
line Returns the current line of the buffer if the buffer is active.
line = {str} Sets the current line of the buffer if the buffer is active.
line_number Returns the number of the current line if the buffer is active.

==
4. Vim::Window objects *ruby-window*

Vim::Window objects represent vim windows.

Class Methods:

current Returns the current window object.
count Returns the number of windows.
self[{n}] Returns the window object for the number {n}. The first number

```
                is 0.

Methods:

buffer          Returns the buffer displayed in the window.
height          Returns the height of the window.
height = {n}    Sets the window height to {n}.
width           Returns the width of the window.
width = {n}     Sets the window width to {n}.
cursor          Returns a [row, col] array for the cursor position.
cursor = [{row}, {col}]
                Sets the cursor position to {row} and {col}.

==============================================================================
5. Global variables                                        *ruby-globals*

There are two global variables.

$curwin         The current window object.
$curbuf         The current buffer object.

==============================================================================
6. Dynamic loading                                         *ruby-dynamic*

On MS-Windows and Unix the Ruby library can be loaded dynamically.  The
|:version| output then includes |+ruby/dyn|.

This means that Vim will search for the Ruby DLL file or shared library only
when needed.  When you don't use the Ruby interface you don't need it, thus
you can use Vim even though this library file is not on your system.


MS-Windows ~

You need to install the right version of Ruby for this to work.  You can find
the package to download from:
http://rubyinstaller.org/downloads/
Currently that is rubyinstaller-2.2.5.exe

To use the Ruby interface the Ruby DLL must be in your search path.  In a
console window type "path" to see what directories are used.  The 'rubydll'
option can be also used to specify the Ruby DLL.

The name of the DLL must match the Ruby version Vim was compiled with.
Currently the name is "msvcrt-ruby220.dll".  That is for Ruby 2.2.X.  To know
for sure edit "gvim.exe" and search for "ruby\d*.dll\c".

If you want to build Vim with RubyInstaller 1.9 or 2.X using MSVC, you need
some tricks.  See the src/INSTALLpc.txt for detail.


Unix ~

The 'rubydll' option can be used to specify the Ruby shared library file
instead of DYNAMIC_RUBY_DLL file what was specified at compile time.  The
version of the shared library must match the Ruby version Vim was compiled
with.

==============================================================================
 vim:tw=78:ts=8:ft=help:norl:
*debugger.txt*  For Vim version 8.0.  Last change: 2005 Mar 29
```

VIM REFERENCE MANUAL by Gordon Prieur

Debugger Support Features *debugger-support*

{Vi does not have any of these features}

==

1. Debugger Features *debugger-features*

The following features are available for an integration with a debugger or an Integrated Programming Environment (IPE) or Integrated Development Environment (IDE):

Alternate Command Input	|alt-input|
Debug Signs	|debug-signs|
Debug Source Highlight	|debug-highlight|
Message Footer	|gui-footer|
Balloon Evaluation	|balloon-eval|

These features were added specifically for use in the Motif version of gvim. However, the |alt-input| and |debug-highlight| were written to be usable in both vim and gvim. Some of the other features could be used in the non-GUI vim with slight modifications. However, I did not do this nor did I test the reliability of building for vim or non Motif GUI versions.

1.1 Alternate Command Input *alt-input*

For Vim to work with a debugger there must be at least an input connection with a debugger or external tool. In many cases there will also be an output connection but this isn't absolutely necessary.

The purpose of the input connection is to let the external debugger send commands to Vim. The commands sent by the debugger should give the debugger enough control to display the current debug environment and state.

The current implementation is based on the X Toolkit dispatch loop and the XtAddInput() function call.

1.2 Debug Signs *debug-signs*

Many debuggers mark specific lines by placing a small sign or color highlight on the line. The |:sign| command lets the debugger set this graphic mark. Some examples where this feature would be used would be a debugger showing an arrow representing the Program Counter (PC) of the program being debugged. Another example would be a small stop sign for a line with a breakpoint. These visible highlights let the user keep track of certain parts of the state of the debugger.

This feature can be used with more than debuggers, too. An IPE can use a sign to highlight build errors, searched text, or other things. The sign feature can also work together with the |debug-highlight| to ensure the mark is highly visible.

Debug signs are defined and placed using the |:sign| command.

1.3 Debug Source Highlight *debug-highlight*

This feature allows a line to have a predominant highlight. The highlight is intended to make a specific line stand out. The highlight could be made to work for both vim and gvim, whereas the debug sign is, in most cases, limited to gvim. The one exception to this is Sun Microsystem's dtterm. The dtterm from Sun has a "sign gutter" for showing signs.

1.4 Message Footer *gui-footer*

The message footer can be used to display messages from a debugger or IPE. It can also be used to display menu and toolbar tips. The footer area is at the bottom of the GUI window, below the line used to display colon commands.

The display of the footer is controlled by the 'guioptions' letter 'F'.

1.5 Balloon Evaluation *balloon-eval*

This feature allows a debugger, or other external tool, to display dynamic information based on where the mouse is pointing. The purpose of this feature was to allow Sun's Visual WorkShop debugger to display expression evaluations. However, the feature was implemented in as general a manner as possible and could be used for displaying other information as well.

The Balloon Evaluation has some settable parameters too. For Motif the font list and colors can be set via X resources (XmNballoonEvalFontList, XmNballoonEvalBackground, and XmNballoonEvalForeground).
The 'balloondelay' option sets the delay before an attempt is made to show a balloon.
The 'ballooneval' option needs to be set to switch it on.

Balloon evaluation is only available when compiled with the |+balloon_eval| feature.

The Balloon evaluation functions are also used to show a tooltip for the toolbar. The 'ballooneval' option does not need to be set for this. But the other settings apply.

Another way to use the balloon is with the 'balloonexpr' option. This is completely user definable.

==
2. Vim Compile Options *debugger-compilation*

The debugger features were added explicitly for use with Sun's Visual WorkShop Integrated Programming Environment (ipe). However, they were done in as generic a manner as possible so that integration with other debuggers could also use some or all of the tools used with Sun's ipe.

The following compile time preprocessor variables control the features:

Alternate Command Input	ALT_X_INPUT
Debug Glyphs	FEAT_SIGNS
Debug Highlights	FEAT_SIGNS
Message Footer	FEAT_FOOTER
Balloon Evaluation	FEAT_BEVAL

The first integration with a full IPE/IDE was with Sun Visual WorkShop. To

compile a gvim which interfaces with VWS set the following flag, which sets
all the above flags:

    Sun Visual WorkShop   FEAT_SUN_WORKSHOP

==
3. Integrated Debuggers   *debugger-integration*

One fully integrated debugger/IPE/IDE is Sun's Visual WorkShop Integrated
Programming Environment.

For Sun NetBeans support see |netbeans|.

 vim:tw=78:sw=4:ts=8:ft=help:norl:
workshop.txt For Vim version 8.0. Last change: 2013 Jul 06

                  VIM REFERENCE MANUAL   by Gordon Prieur

Sun Visual WorkShop Features   *workshop* *workshop-support*

1. Introduction |workshop-intro|
2. Commands |workshop-commands|
3. Compiling vim/gvim for WorkShop |workshop-compiling|
4. Configuring gvim for a WorkShop release tree |workshop-configure|
5. Obtaining the latest version of the XPM library |workshop-xpm|

{Vi does not have any of these features}
{only available when compiled with the |+sun_workshop| feature}

==
1. Introduction   *workshop-intro*

Sun Visual WorkShop has an "Editor of Choice" feature designed to let users
debug using their favorite editors. For the 6.0 release we have added support
for gvim. A workshop debug session will have a debugging window and an editor
window (possibly others as well). The user can do many debugging operations
from the editor window, minimizing the need to switch from window to window.

The version of vim shipped with Sun Visual WorkShop 6 (also called Forte
Developer 6) is vim 5.3. The features in this release are much more reliable
than the vim/gvim shipped with Visual WorkShop. VWS users wishing to use vim
as their editor should compile these sources and install them in their
workshop release tree.

==
2. Commands   *workshop-commands*

                                                :ws *:wsverb*
:ws[verb] verb   Pass the verb to the verb executor

Pass the verb to a workshop function which gathers some arguments and
sends the verb and data to workshop over an IPC connection.

==
3. Compiling vim/gvim for WorkShop   *workshop-compiling*

Compiling vim with FEAT_SUN_WORKSHOP turns on all compile time flags necessary
for building a vim to work with Visual WorkShop. The features required for VWS
have been built and tested using the Sun compilers from the VWS release. They
have not been built or tested using Gnu compilers. This does not mean the

features won't build and run if compiled with gcc, just that nothing is
guaranteed with gcc!

==
4. Configuring gvim for a WorkShop release tree   *workshop-configure*

There are several assumptions which must be met in order to compile a gvim for
use with Sun Visual WorkShop 6.

    o You should use the compiler in VWS rather than gcc. We have neither
      built nor tested with gcc and cannot guarantee it will build properly.

    o You must supply your own XPM library. See |workshop-xpm| below for
      details on obtaining the latest version of XPM.

    o Edit the Makefile in the src directory and uncomment the lines for Sun
      Visual WorkShop. You can easily find these by searching for the string
      FEAT_SUN_WORKSHOP

    o We also suggest you use Motif for your gui. This will provide gvim with
      the same look-and-feel as the rest of Sun Visual WorkShop.

The following configuration line can be used to configure vim to build for use
with Sun Visual WorkShop:

    $ CC=cc configure --enable-workshop --enable-gui=motif \
        -prefix=<VWS-install-dir>/contrib/contrib6/<vim-version>

The VWS-install-dir should be the base directory where your Sun Visual WorkShop
was installed. By default this is /opt/SUNWspro. It will normally require
root permissions to install the vim release. You will also need to change the
symlink <VWS-install-dir>/bin/gvim to point to the vim in your newly installed
directory. The <vim-version> should be a unique version string. I use "vim"
concatenated with the equivalent of version.h's VIM_VERSION_SHORT.

==
5. Obtaining the latest version of the XPM library   *workshop-xpm*

The XPM library is required to show images within Vim with Motif or Athena.
Without it the toolbar and signs will be disabled.

The XPM library is provided by Arnaud Le Hors of the French National Institute
for Research in Computer Science and Control. It can be downloaded from
http://cgit.freedesktop.org/xorg/lib/libXpm. The current release, as of this
writing, is xpm-3.4k-solaris.tgz, which is a gzip'ed tar file. If you create
the directory /usr/local/xpm and untar the file there you can use the
uncommented lines in the Makefile without changing them. If you use another
xpm directory you will need to change the XPM_DIR in src/Makefile.

 vim:tw=78:ts=8:ft=help:norl:
netbeans.txt For Vim version 8.0. Last change: 2016 Jul 15

                  VIM REFERENCE MANUAL   by Gordon Prieur et al.

                                      netbeans *netbeans-support*

Vim NetBeans Protocol: a socket interface for Vim integration into an IDE.

{Vi does not have any of these features}
{only available when compiled with the |+netbeans_intg| feature}

==
1. Introduction   *netbeans-intro*

The NetBeans interface was initially developed to integrate Vim into the
NetBeans Java IDE, using the external editor plugin. This NetBeans plugin no
longer exists for recent versions of NetBeans but the protocol was developed
in such a way that any IDE can use it to integrate Vim.

The NetBeans protocol of Vim is a text based communication protocol, over a
classical TCP socket. There is no dependency on Java or NetBeans. Any language
or environment providing a socket interface can control Vim using this
protocol. There are existing implementations in C, C++, Python and Java. The
name NetBeans is kept today for historical reasons.

Current projects using the NetBeans protocol of Vim are:
- VimIntegration, description of various projects doing Vim Integration:
        http://www.freehackers.org/VimIntegration
- Agide, an IDE for the AAP project, written in Python:
        http://www.a-a-p.org
- Clewn, a gdb integration into Vim, written in C:
        http://clewn.sourceforge.net/
- Pyclewn, a gdb integration into Vim, written in Python:
        http://pyclewn.sourceforge.net/
- VimPlugin, integration of Vim inside Eclipse:
        http://vimplugin.sourceforge.net/wiki/pmwiki.php
- PIDA, IDE written in Python integrating Vim:
        http://pida.co.uk/
- VimWrapper, library to easy Vim integration into IDE:
        http://www.freehackers.org/VimWrapper

Check the specific project pages to see how to use Vim with these projects.

An alternative is to use a channel, see |channel|.

In the rest of this help page, we will use the term "Vim Controller" to
describe the program controlling Vim through the NetBeans socket interface.

About the NetBeans IDE ~

NetBeans is an open source Integrated Development Environment developed
jointly by Sun Microsystems, Inc. and the netbeans.org developer community.
Initially just a Java IDE, NetBeans has had C, C++, and Fortran support added
in recent releases.

For more information visit the main NetBeans web site http://www.netbeans.org.
The External Editor is now, unfortunately, declared obsolete. See
 http://externaleditor.netbeans.org.

Sun Microsystems, Inc. also ships NetBeans under the name Sun ONE Studio.
Visit http://www.sun.com for more information regarding the Sun ONE Studio
product line.

Current releases of NetBeans provide full support for Java and limited support
for C, C++, and Fortran. Current releases of Sun ONE Studio provide full
support for Java, C, C++, and Fortran.

==
2. Integration features   *netbeans-integration*

The NetBeans socket interface of Vim allows to get information from Vim or to
ask Vim to perform specific actions:
- get information about buffer: buffer name, cursor position, buffer content,
  etc.
- be notified when buffers are open or closed
- be notified of how the buffer content is modified
- load and save files
- modify the buffer content
- installing special key bindings
- raise the window, control the window geometry

For sending key strokes to Vim or for evaluating functions in Vim, you must
use the |clientserver| interface.

==
3. Configuring Vim for NetBeans   *netbeans-configure*

For more help about installing Vim, please read |usr_90.txt| in the Vim User
Manual.

On Unix:

When running configure without arguments the NetBeans interface should be
included. That is, if the configure check to find out if your system supports
the required features succeeds.

In case you do not want the NetBeans interface you can disable it by
uncommenting a line with "--disable-netbeans" in the Makefile.

Currently the NetBeans interface is supported by Vim running in a terminal and
by gvim when it is run with one of the following GUIs: GTK, GNOME, Windows,
Athena and Motif.

If Motif support is required the user must supply XPM libraries. See
|workshop-xpm| for details on obtaining the latest version of XPM.

On MS-Windows:

The Win32 support is now in beta stage.

To use XPM signs on Win32 (e.g. when using with NetBeans) you can compile
XPM by yourself or use precompiled libraries from http://iamphet.nm.ru/misc/

(for MS Visual C++) or http://gnuwin32.sourceforge.net (for MinGW).

Enable debugging:

To enable debugging of Vim and of the NetBeans protocol, the "NBDEBUG" macro needs to be defined. Search in the Makefile of the platform you are using for "NBDEBUG" to see what line needs to be uncommented. This effectively adds "-DNBDEBUG" to the compile command. Also see |netbeans-debugging|

==
4. Error Messages *netbeans-messages*

These error messages are specific to NetBeans socket protocol:

E463
Region is guarded, cannot modify
The Vim Controller has defined guarded areas in the text, which you cannot change. Also sets the current buffer, if necessary.

E532
The defineAnnoType highlighting color name is too long
The maximum length of the "fg" or "bg" color argument in the defineAnnoType command is 32 characters.
New in version 2.5.

E656
Writes of unmodified buffers forbidden
Writes of unmodified buffers that were opened from the Vim Controller are not possible.

E657
Partial writes disallowed
Partial writes for buffers that were opened from the Vim Controller are not allowed.

E658
Connection lost for this buffer
The Vim Controller has become confused about the state of this file. Rather than risk data corruption, it has severed the connection for this file. Vim will take over responsibility for saving changes to this file and the Vim Controller will no longer know of these changes.

E744
Read-only file
Vim normally allows changes to a read-only file and only enforces the read-only rule if you try to write the file. However, NetBeans does not let you make changes to a file which is read-only and becomes confused if Vim does this. So Vim does not allow modifications to files when run in NetBeans mode.

==
5. Running Vim in NetBeans mode *netbeans-run*

There are two different ways to run Vim in NetBeans mode:

+ an IDE may start Vim with the |-nb| command line argument
+ NetBeans can be started from within Vim with the |:nbstart| command

```
Vim uses a 3 second timeout on trying to make the connection.

                                                        *netbeans-parameters*
Three forms can be used to setup the NetBeans connection parameters.
When started from the command line, the |-nb| command line argument may be:

        -nb={fname}                                 from a file
        -nb:{hostname}:{addr}:{password}            directly
        -nb                                         from a file or environment

When started from within Vim, the |:nbstart| optional argument may be:

        ={fname}                                    from a file
        :{hostname}:{addr}:{password}               directly
        <MISSING ARGUMENT>                          from a file or environment

                                                        *E660* *E668*
When NetBeans is started from the command line, for security reasons, the best
method is to write the information in a file readable only by the user.  The
name of the file can be passed with the "-nb={fname}" argument or, when "-nb"
is used without a parameter, the environment variable "__NETBEANS_CONINFO".
The file must contain these three lines, in any order:

        host={hostname}
        port={addr}
        auth={password}

Other lines are ignored.  The Vim Controller is responsible for deleting the
file afterwards.

{hostname} is the name of the machine where Vim Controller is running.  When
omitted the environment variable "__NETBEANS_HOST" is used or the default
"localhost".

{addr} is the port number for the NetBeans interface.  When omitted the
environment variable "__NETBEANS_SOCKET" is used or the default 3219.

{password} is the password for connecting to NetBeans.  When omitted the
environment variable "__NETBEANS_VIM_PASSWORD" is used or "changeme".

Vim will initiate a socket connection (client side) to the specified host and
port upon startup. The password will be sent with the AUTH event when the
connection has been established.


==============================================================================
6. NetBeans protocol                                    *netbeans-protocol*

The communication between the Vim Controller and Vim uses plain text
messages.  This protocol was first designed to work with the external editor
module of NetBeans.  Later it was extended to work with Agide (A-A-P GUI IDE,
see http://www.a-a-p.org) and then with other IDE. The extensions are marked
with "version 2.1".

Version 2.2 of the protocol has several minor changes which should only affect
NetBeans users (ie, not Agide users).  However, a bug was fixed which could
cause confusion.  The netbeans_saved() function sent a "save" protocol
command.  In protocol version 2.1 and earlier this was incorrectly interpreted
as a notification that a write had taken place.  In reality, it told NetBeans
to save the file so multiple writes were being done.  This caused various
problems and has been fixed in 2.2.  To decrease the likelihood of this
confusion happening again, netbeans_saved() has been renamed to
```

netbeans_save_buffer().

We are now at version 2.5. For the differences between 2.4 and 2.5 search for "2.5" below.

The messages are currently sent over a socket. Since the messages are in plain UTF-8 text this protocol could also be used with any other communication mechanism.

Netbeans messages are processed when Vim is idle, waiting for user input. When Vim is run in non-interactive mode, for example when running an automated test case that sources a Vim script, the idle loop may not be called often enough. In that case, insert |:sleep| commands in the Vim script. The |:sleep| command does invoke Netbeans messages processing.

6.1 Kinds of messages *nb-messages*

There are four kinds of messages:

kind	direction	comment ~
Command	IDE -> editor	no reply necessary
Function	IDE -> editor	editor must send back a reply
Reply	editor -> IDE	only in response to a Function
Event	editor -> IDE	no reply necessary

The messages are sent as a single line with a terminating newline character. Arguments are separated by a single space. The first item of the message depends on the kind of message:

kind	first item	example ~
Command	bufID:name!seqno	11:showBalloon!123 "text"
Function	bufID:name/seqno	11:getLength/123
Reply	seqno	123 5000
Event	bufID:name=seqno	11:keyCommand=123 "S-F2"

6.2 Terms *nb-terms*

bufID Buffer number. A message may be either for a specific buffer or generic. Generic messages use a bufID of zero. NOTE: this buffer ID is assigned by the IDE, it is not Vim's buffer number. The bufID must be a sequentially rising number, starting at one. When the 'switchbuf' option is set to "usetab" and the "bufID" buffer is not found in the current tab page, the netbeans commands and functions that set this buffer as the current buffer will jump to the first open window that contains this buffer in other tab pages instead of replacing the buffer in the current window.

seqno The IDE uses a sequence number for Commands and Functions. A Reply must use the sequence number of the Function that it is associated with. A zero sequence number can be used for

```
                Events (the seqno of the last received Command or Function can
                also be used).

string          Argument in double quotes.  Text is in UTF-8 encoding.  This
                means ASCII is passed as-is.  Special characters are
                represented with a backslash:
                        \"      double quote
                        \n      newline
                        \r      carriage-return
                        \t      tab (optional, also works literally)
                        \\      backslash
                NUL bytes are not allowed!

boolean         Argument with two possible values:
                        T       true
                        F       false

number          Argument with a decimal number.

color           Argument with either a decimal number, "none" (without the
                quotes) or the name of a color (without the quotes) defined
                both in the color list in |highlight-ctermfg| and in the color
                list in |gui-colors|.
                New in version 2.5.

offset          A number argument that indicates a byte position in a buffer.
                The first byte has offset zero.  Line breaks are counted for
                how they appear in the file (CR/LF counts for two bytes).
                Note that a multi-byte character is counted for the number of
                bytes it takes.

lnum/col        Argument with a line number and column number position.  The
                line number starts with one, the column is the byte position,
                starting with zero.  Note that a multi-byte character counts
                for several columns.

pathname        String argument: file name with full path.


6.3 Commands                                                *nb-commands*

actionMenuItem  Not implemented.

actionSensitivity
                Not implemented.

addAnno serNum typeNum off len
                Place an annotation in this buffer.
                Arguments:
                   serNum       number  serial number of this placed
                                        annotation, used to be able to remove
                                        it
                   typeNum      number  sequence number of the annotation
                                        defined with defineAnnoType for this
                                        buffer
                   off          number  offset where annotation is to be placed
                   len          number  not used
                In version 2.1 "lnum/col" can be used instead of "off".

balloonResult text
                Not implemented.
```

```
close           Close the buffer.  This leaves us without current buffer, very
                dangerous to use!

create          Creates a buffer without a name.  Replaces the current buffer
                (it's hidden when it was changed).
                The Vim Controller should use this as the first command for a
                file that is being opened.  The sequence of commands could be:
                        create
                        setCaretListener        (ignored)
                        setModified             (no effect)
                        setContentType          (ignored)
                        startDocumentListen
                        setTitle
                        setFullName

defineAnnoType typeNum typeName tooltip glyphFile fg bg
                Define a type of annotation for this buffer.
                Arguments:
                   typeNum      number  sequence number (not really used)
                   typeName     string  name that identifies this annotation
                   tooltip      string  not used
                   glyphFile    string  name of icon file
                   fg           color   foreground color for line highlighting
                   bg           color   background color for line highlighting
                Vim will define a sign for the annotation.
                When color is a number, this is the "#rrggbb" Red, Green and
                Blue values of the color (see |gui-colors|) and the
                highlighting is only defined for GVim.
                When color is a name, this color is defined both for Vim
                running in a color terminal and for GVim.
                When both "fg" and "bg" are "none" no line highlighting is
                used (new in version 2.1).
                When "glyphFile" is empty, no text sign is used (new in
                version 2.1).
                When "glyphFile" is one or two characters long, a text sign is
                defined (new in version 2.1).
                Note: the annotations will be defined in sequence, and the
                sequence number is later used with addAnno.

editFile pathname
                Set the name for the buffer and edit the file "pathname", a
                string argument.
                Normal way for the IDE to tell the editor to edit a file.

                You must set a bufId different of 0 with this command to
                assign a bufId to the buffer. It will trigger an event
                fileOpened with a bufId of 0 but the buffer has been assigned.

                If the IDE is going to pass the file text to the editor use
                these commands instead:
                        setFullName
                        insert
                        initDone
                New in version 2.1.

enableBalloonEval
                Not implemented.

endAtomic       End an atomic operation.  The changes between "startAtomic"
                and "endAtomic" can be undone as one operation.  But it's not
                implemented yet.  Redraw when necessary.
```

```
guard off len
                Mark an area in the buffer as guarded.  This means it cannot
                be edited.  "off" and "len" are numbers and specify the text
                to be guarded.

initDone        Mark the buffer as ready for use.  Implicitly makes the buffer
                the current buffer.  Fires the BufReadPost autocommand event.

insertDone
                Sent by Vim Controller to tell Vim an initial file insert is
                done.  This triggers a read message being printed.  Prior to
                version 2.3, no read messages were displayed after opening a
                file.  New in version 2.3.

moveAnnoToFront serNum
                Not implemented.

netbeansBuffer isNetbeansBuffer
                If "isNetbeansBuffer" is "T" then this buffer is "owned" by
                NetBeans.
                New in version 2.2.

putBufferNumber pathname
                Associate a buffer number with the Vim buffer by the name
                "pathname", a string argument.  To be used when the editor
                reported editing another file to the IDE and the IDE needs to
                tell the editor what buffer number it will use for this file.
                Also marks the buffer as initialized.
                New in version 2.1.

raise           Bring the editor to the foreground.
                Only when Vim is run with a GUI.
                New in version 2.1.

removeAnno serNum
                Remove a previously placed annotation for this buffer.
                "serNum" is the same number used in addAnno.

save            Save the buffer when it was modified.  The other side of the
                interface is expected to write the buffer and invoke
                "setModified" to reset the "changed" flag of the buffer.
                The writing is skipped when one of these conditions is true:
                - 'write' is not set
                - the buffer is read-only
                - the buffer does not have a file name
                - 'buftype' disallows writing
                New in version 2.2.

saveDone
                Sent by Vim Controller to tell Vim a save is done.  This
                triggers a save message being printed.  Prior to version 2.3,
                no save messages were displayed after a save.
                New in version 2.3.

setAsUser       Not implemented.

setBufferNumber pathname
                Associate a buffer number with Vim buffer by the name
                "pathname".  To be used when the editor reported editing
                another file to the IDE and the IDE needs to tell the editor
                what buffer number it will use for this file.
                Has the side effect of making the buffer the current buffer.
```

```
                See "putBufferNumber" for a more useful command.

setContentType
                Not implemented.

setDot off      Make the buffer the current buffer and set the cursor at the
                specified position.  If the buffer is open in another window
                than make that window the current window.
                If there are folds they are opened to make the cursor line
                visible.
                In version 2.1 "lnum/col" can be used instead of "off".

setExitDelay seconds
                Set the delay for exiting to "seconds", a number.
                This delay is used to give the IDE a chance to handle things
                before really exiting.  The default delay is two seconds.
                New in version 2.1.
                Obsolete in version 2.3.

setFullName pathname
                Set the file name to be used for a buffer to "pathname", a
                string argument.
                Used when the IDE wants to edit a file under control of the
                IDE.  This makes the buffer the current buffer, but does not
                read the file.  "insert" commands will be used next to set the
                contents.

setLocAndSize   Not implemented.

setMark         Not implemented.

setModified modified
                When the boolean argument "modified" is "T" mark the buffer as
                modified, when it is "F" mark it as unmodified.

setModtime time
                Update a buffers modification time after the file has been
                saved directly by the Vim Controller.
                New in version 2.3.

setReadOnly
                Set a file as readonly
                Implemented in version 2.3.

setStyle        Not implemented.

setTitle name
                Set the title for the buffer to "name", a string argument.
                The title is only used for the Vim Controller functions, not
                by Vim.

setVisible visible
                When the boolean argument "visible" is "T", goto the buffer.
                The "F" argument does nothing.

showBalloon text
                Show a balloon (popup window) at the mouse pointer position,
                containing "text", a string argument.  The balloon should
                disappear when the mouse is moved more than a few pixels.
                Only when Vim is run with a GUI.
                New in version 2.1.
```

```
specialKeys
                Map a set of keys (mostly function keys) to be passed back
                to the Vim Controller for processing.  This lets regular IDE
                hotkeys be used from Vim.
                Implemented in version 2.3.

startAtomic     Begin an atomic operation.  The screen will not be updated
                until "endAtomic" is given.

startCaretListen
                Not implemented.

startDocumentListen
                Mark the buffer to report changes to the IDE with the
                "insert" and "remove" events.  The default is to report
                changes.

stopCaretListen
                Not implemented.

stopDocumentListen
                Mark the buffer to stop reporting changes to the IDE.
                Opposite of startDocumentListen.
                NOTE: if "netbeansBuffer" was used to mark this buffer as a
                NetBeans buffer, then the buffer is deleted in Vim.  This is
                for compatibility with Sun Studio 10.

unguard off len
                Opposite of "guard", remove guarding for a text area.
                Also sets the current buffer, if necessary.

version         Not implemented.


6.4 Functions and Replies                               *nb-functions*

getDot          Not implemented.

getCursor       Return the current buffer and cursor position.
                The reply is:
                        seqno bufID lnum col off
                seqno = sequence number of the function
                bufID = buffer ID of the current buffer (if this is unknown -1
                        is used)
                lnum  = line number of the cursor (first line is one)
                col   = column number of the cursor (in bytes, zero based)
                off   = offset of the cursor in the buffer (in bytes)
                New in version 2.1.

getLength       Return the length of the buffer in bytes.
                Reply example for a buffer with 5000 bytes:
                        123 5000
                TODO: explain use of partial line.

getMark         Not implemented.

getAnno serNum
                Return the line number of the annotation in the buffer.
                Argument:
                        serNum          serial number of this placed annotation
                The reply is:
                        123 lnum        line number of the annotation
```

```
                        123 0           invalid annotation serial number
                New in version 2.4.

getModified     When a buffer is specified: Return zero if the buffer does not
                have changes, one if it does have changes.
                When no buffer is specified (buffer number zero): Return the
                number of buffers with changes.  When the result is zero it's
                safe to tell Vim to exit.
                New in version 2.1.

getText         Return the contents of the buffer as a string.
                Reply example for a buffer with two lines
                        123 "first line\nsecond line\n"
                NOTE: docs indicate an offset and length argument, but this is
                not implemented.

insert off text
                Insert "text" before position "off".  "text" is a string
                argument, "off" a number.
                "text" should have a "\n" (newline) at the end of each line.
                Or "\r\n" when 'fileformat' is "dos".  When using "insert" in
                an empty buffer Vim will set 'fileformat' accordingly.
                When "off" points to the start of a line the text is inserted
                above this line.  Thus when "off" is zero lines are inserted
                before the first line.
                When "off" points after the start of a line, possibly on the
                NUL at the end of a line, the first line of text is appended
                to this line.  Further lines come below it.
                Possible replies:
                        123             no problem
                        123 !message    failed
                Note that the message in the reply is not quoted.
                Also sets the current buffer, if necessary.
                Does not move the cursor to the changed text.
                Resets undo information.

remove off length
                Delete "length" bytes of text at position "off".  Both
                arguments are numbers.
                Possible replies:
                        123             no problem
                        123 !message    failed
                Note that the message in the reply is not quoted.
                Also sets the current buffer, if necessary.

saveAndExit     Perform the equivalent of closing Vim: ":confirm qall".
                If there are no changed files or the user does not cancel the
                operation Vim exits and no result is sent back.  The IDE can
                consider closing the connection as a successful result.
                If the user cancels the operation the number of modified
                buffers that remains is returned and Vim does not exit.
                New in version 2.1.


6.5 Events                                              *nb-events*

balloonEval off len type
                The mouse pointer rests on text for a short while.  When "len"
                is zero, there is no selection and the pointer is at position
                "off".  When "len" is non-zero the text from position "off" to
                "off" + "len" is selected.
                Only sent after "enableBalloonEval" was used for this buffer.
```

```
                "type" is not yet defined.
                Not implemented yet.

balloonText text
                Used when 'ballooneval' is set and the mouse pointer rests on
                some text for a moment.  "text" is a string, the text under
                the mouse pointer.
                Only when Vim is run with a GUI.
                New in version 2.1.

buttonRelease button lnum col
                Report which button was pressed and the location of the cursor
                at the time of the release.  Only for buffers that are owned
                by the Vim Controller.  This event is not sent if the button
                was released while the mouse was in the status line or in a
                separator line.  If col is less than 1 the button release was
                in the sign area.
                New in version 2.2.

disconnect
                Tell the Vim Controller that Vim is exiting and not to try and
                read or write more commands.
                New in version 2.3.

fileClosed      Not implemented.

fileModified    Not implemented.

fileOpened pathname open modified
                A file was opened by the user.
                Arguments:
                   pathname     string    name of the file
                   open         boolean   always "T"
                   modified     boolean   always "F"

geometry cols rows x y
                Report the size and position of the editor window.
                Arguments:
                   cols         number    number of text columns
                   rows         number    number of text rows
                   x            number    pixel position on screen
                   y            number    pixel position on screen
                Only works for Motif.

insert off text
                Text "text" has been inserted in Vim at position "off".
                Only fired when enabled, see "startDocumentListen".

invokeAction    Not implemented.

keyCommand keyName
                Reports a special key being pressed with name "keyName", which
                is a string.
                Supported key names:
                        F1              function key 1
                        F2              function key 2
                        ...
                        F12             function key 12

                        ' '             space (without the quotes)
                        !               exclamation mark
                        ...             any other ASCII printable character
```

```
                        ~               tilde

                        X               any unrecognized key

                The key may be prepended by "C", "S" and/or "M" for Control,
                Shift and Meta (Alt) modifiers.  If there is a modifier a dash
                is used to separate it from the key name.  For example:
                "C-F2".
                ASCII characters are new in version 2.1.

keyAtPos keyName lnum/col
                Like "keyCommand" and also report the line number and column
                of the cursor.
                New in version 2.1.

killed          A file was deleted or wiped out by the user and the buffer
                annotations have been removed.  The bufID number for this
                buffer has become invalid.  Only for files that have been
                assigned a bufID number by the IDE.

newDotAndMark off off
                Reports the position of the cursor being at "off" bytes into
                the buffer.  Only sent just before a "keyCommand" event.

quit            Not implemented.

remove off len
                Text was deleted in Vim at position "off" with byte length
                "len".
                Only fired when enabled, see "startDocumentListen".

revert          Not implemented.

save            The buffer has been saved and is now unmodified.
                Only fired when enabled, see "startDocumentListen".

startupDone     The editor has finished its startup work and is ready for
                editing files.
                New in version 2.1.

unmodified      The buffer is now unmodified.
                Only fired when enabled, see "startDocumentListen".

version vers    Report the version of the interface implementation.  Vim
                reports "2.4" (including the quotes).


6.6 Special messages                                    *nb-special*

These messages do not follow the style of the messages above.  They are
terminated by a newline character.

ACCEPT          Not used.

AUTH password   editor -> IDE: First message that the editor sends to the IDE.
                Must contain the password for the socket server, as specified
                with the |-nb| argument.  No quotes are used!

DISCONNECT      IDE -> editor: break the connection.  The editor will exit.
                The IDE must only send this message when there are no unsaved
                changes!
```

```
DETACH          IDE -> editor: break the connection without exiting the
                editor.  Used when the IDE exits without bringing down the
                editor as well.
                New in version 2.1.

REJECT          Not used.


6.7 Protocol errors                                     *nb-protocol_errors*

These errors occur when a message violates the protocol:
*E627* *E628* *E629* *E632* *E633* *E634* *E635* *E636*
*E637* *E638* *E639* *E640* *E641* *E642* *E643* *E644* *E645* *E646*
*E647* *E648* *E649* *E650* *E651* *E652*


==============================================================================
7. NetBeans commands                                    *netbeans-commands*

                                                        *:nbstart* *E511* *E838*
:nbs[tart] {connection} Start a new Netbeans session with {connection} as the
                        socket connection parameters.  The format of
                        {connection} is described in |netbeans-parameters|.
                        At any time, one may check if the netbeans socket is
                        connected by running the command:
                        ':echo has("netbeans_enabled")'

                                                        *:nbclose*
:nbc[lose]              Close the current NetBeans session. Remove all placed
                        signs.

                                                        *:nbkey*
:nb[key] {key}          Pass the {key} to the Vim Controller for processing.
                        When a hot-key has been installed with the specialKeys
                        command, this command can be used to generate a hotkey
                        message to the Vim Controller.
                        This command can also be used to pass any text to the
                        Vim  Controller. It is used by Pyclewn, for example,
                        to build the complete set of gdb commands as Vim user
                        commands.
                        The events newDotAndMark, keyCommand and keyAtPos are
                        generated (in this order).


==============================================================================
8. Known problems                                       *netbeans-problems*

NUL bytes are not possible.  For editor -> IDE they will appear as NL
characters.  For IDE -> editor they cannot be inserted.

A NetBeans session may be initiated with Vim running in a terminal, and
continued later in a GUI environment after running the |:gui| command. In this
case, the highlighting defined for the NetBeans annotations may be cleared
when the ":gui" command sources .gvimrc and this file loads a colorscheme
that runs the command ":highlight clear".
New in version 2.5.


==============================================================================
9. Debugging NetBeans protocol                          *netbeans-debugging*

To debug the Vim protocol, you must first compile Vim with debugging support
```

and NetBeans debugging support. See |netbeans-configure| for instructions
about Vim compiling and how to enable debug support.

When running Vim, set the following environment variables:

```
    export SPRO_GVIM_DEBUG=netbeans.log
    export SPRO_GVIM_DLEVEL=0xffffffff
```

Vim will then log all the incoming and outgoing messages of the NetBeans
protocol to the file netbeans.log .

The content of netbeans.log after a session looks like this:

```
Tue May 20 17:19:27 2008
EVT: 0:startupDone=0
CMD 1: (1) create
CMD 2: (1) setTitle "testfile1.txt"
CMD 3: (1) setFullName "testfile1.txt"
EVT(suppressed): 1:remove=3 0 -1
EVT: 1:fileOpened=0 "d:\\work\\vimWrapper\\vimWrapper2\\pyvimwrapper\\tests\
\testfile1.txt" T F
CMD 4: (1) initDone
FUN 5: (0) getCursor
REP 5: 1 1 0 0
CMD 6: (2) create
CMD 7: (2) setTitle "testfile2.txt"
CMD 8: (2) setFullName "testfile2.txt"
EVT(suppressed): 2:remove=8 0 -1
EVT: 2:fileOpened=0 "d:\\work\\vimWrapper\\vimWrapper2\\pyvimwrapper\\tests\
\testfile2.txt" T F
CMD 9: (2) initDone
```

==
10. NetBeans External Editor

NOTE: This information is obsolete! Only relevant if you are using an old
version of NetBeans.

10.1. Downloading NetBeans *netbeans-download*

The NetBeans IDE is available for download from netbeans.org. You can download
a released version, download sources, or use CVS to download the current
source tree. If you choose to download sources, follow directions from
netbeans.org on building NetBeans.

Depending on the version of NetBeans you download, you may need to do further
work to get the required External Editor module. This is the module which lets
NetBeans work with gvim (or xemacs :-). See http://externaleditor.netbeans.org
for details on downloading this module if your NetBeans release does not have
it.

For C, C++, and Fortran support you will also need the cpp module. See
http://cpp.netbeans.org for information regarding this module.

You can also download Sun ONE Studio from Sun Microsystems, Inc for a 30 day
free trial. See http://www.sun.com for further details.

10.2. NetBeans Key Bindings *netbeans-keybindings*

Vim understands a number of key bindings that execute NetBeans commands.

These are typically all the Function key combinations. To execute a NetBeans command, the user must press the Pause key followed by a NetBeans key binding. For example, in order to compile a Java file, the NetBeans key binding is "F9". So, while in vim, press "Pause F9" to compile a java file. To toggle a breakpoint at the current line, press "Pause Shift F8".

The Pause key is Function key 21. If you don't have a working Pause key and want to use F8 instead, use: >

```
:map <F8> <F21>
```

The External Editor module dynamically reads the NetBeans key bindings so vim should always have the latest key bindings, even when NetBeans changes them.

10.3. Preparing NetBeans for Vim *netbeans-preparation*

In order for NetBeans to work with vim, the NetBeans External Editor module must be loaded and enabled. If you have a Sun ONE Studio Enterprise Edition then this module should be loaded and enabled. If you have a NetBeans release you may need to find another way of obtaining this open source module.

You can check if you have this module by opening the Tools->Options dialog and drilling down to the "Modules" list (IDE Configuration->System->Modules). If your Modules list has an entry for "External Editor" you must make sure it is enabled (the "Enabled" property should have the value "True"). If your Modules list has no External Editor see the next section on |obtaining-exted|.

10.4. Obtaining the External Editor Module *obtaining-exted*

There are 2 ways of obtaining the External Editor module. The easiest way is to use the NetBeans Update Center to download and install the module. Unfortunately, some versions do not have this module in their update center. If you cannot download via the update center you will need to download sources and build the module. I will try and get the module available from the NetBeans Update Center so building will be unnecessary. Also check http://externaleditor.netbeans.org for other availability options.

To download the External Editor sources via CVS and build your own module, see http://externaleditor.netbeans.org and http://www.netbeans.org. Unfortunately, this is not a trivial procedure.

10.5. Setting up NetBeans to run with Vim *netbeans-setup*

Assuming you have loaded and enabled the NetBeans External Editor module as described in |netbeans-preparation| all you need to do is verify that the gvim command line is properly configured for your environment.

Open the Tools->Options dialog and open the Editing category. Select the External Editor. The right hand pane should contain a Properties tab and an Expert tab. In the Properties tab make sure the "Editor Type" is set to "Vim". In the Expert tab make sure the "Vim Command" is correct.

You should be careful if you change the "Vim Command". There are command line options there which must be there for the connection to be properly set up. You can change the command name but that's about it. If your gvim can be found by your $PATH then the Vim Command can start with "gvim". If you don't want gvim searched from your $PATH then hard code in the full Unix path name. At this point you should get a gvim for any source file you open in NetBeans.

If some files come up in gvim and others (with different file suffixes) come
up in the default NetBeans editor you should verify the MIME type in the
Expert tab MIME Type property. NetBeans is MIME oriented and the External
Editor will only open MIME types specified in this property.

 vim:tw=78:ts=8:ft=help:norl:
sign.txt   For Vim version 8.0. Last change: 2016 Aug 17

                  VIM REFERENCE MANUAL   by Gordon Prieur
                                          and Bram Moolenaar

Sign Support Features   *sign-support*

1. Introduction |sign-intro|
2. Commands |sign-commands|

{Vi does not have any of these features}
{only available when compiled with the |+signs| feature}

==
1. Introduction   *sign-intro* *signs*

When a debugger or other IDE tool is driving an editor it needs to be able
to give specific highlights which quickly tell the user useful information
about the file. One example of this would be a debugger which had an icon
in the left-hand column denoting a breakpoint. Another example might be an
arrow representing the Program Counter (PC). The sign features allow both
placement of a sign, or icon, in the left-hand side of the window and
definition of a highlight which will be applied to that line. Displaying the
sign as an image is most likely only feasible in gvim (although Sun
Microsystem's dtterm does support this it's the only terminal emulator I know
of which does). A text sign and the highlight should be feasible in any color
terminal emulator.

Signs and highlights are not useful just for debuggers. Sun's Visual
WorkShop uses signs and highlights to mark build errors and SourceBrowser
hits. Additionally, the debugger supports 8 to 10 different signs and
highlight colors. |workshop| Same for Netbeans |netbeans|.

There are two steps in using signs:

1. Define the sign. This specifies the image, text and highlighting. For
   example, you can define a "break" sign with an image of a stop roadsign and
   text "!!".

2. Place the sign. This specifies the file and line number where the sign is
   displayed. A defined sign can be placed several times in different lines
   and files.

When signs are defined for a file, Vim will automatically add a column of two
characters to display them in. When the last sign is unplaced the column
disappears again. This behavior can be changed with the 'signcolumn' option.

The color of the column is set with the SignColumn group |hl-SignColumn|.
Example to set the color: >

        :highlight SignColumn guibg=darkgrey

```
==============================================================================
2. Commands                                        *sign-commands* *:sig* *:sign*

Here is an example that places a sign "piet", displayed with the text ">>", in
line 23 of the current file: >
        :sign define piet text=>> texthl=Search
        :exe ":sign place 2 line=23 name=piet file=" . expand("%:p")

And here is the command to delete it again: >
        :sign unplace 2

Note that the ":sign" command cannot be followed by another command or a
comment.  If you do need that, use the |:execute| command.


DEFINING A SIGN.                          *:sign-define* *E255* *E160* *E612*

:sign define {name} {argument}...
                Define a new sign or set attributes for an existing sign.
                The {name} can either be a number (all digits) or a name
                starting with a non-digit.  Leading digits are ignored, thus
                "0012", "012" and "12" are considered the same name.
                About 120 different signs can be defined.

                Accepted arguments:

        icon={bitmap}
                Define the file name where the bitmap can be found.  Should be
                a full path.  The bitmap should fit in the place of two
                characters.  This is not checked.  If the bitmap is too big it
                will cause redraw problems.  Only GTK 2 can scale the bitmap
                to fit the space available.
                        toolkit         supports ~
                        GTK 1           pixmap (.xpm)
                        GTK 2           many
                        Motif           pixmap (.xpm)
                        Win32           .bmp, .ico, .cur
                                        pixmap (.xpm) |+xpm_w32|

        linehl={group}
                Highlighting group used for the whole line the sign is placed
                in.  Most useful is defining a background color.

        text={text}                                                 *E239*
                Define the text that is displayed when there is no icon or the
                GUI is not being used.  Only printable characters are allowed
                and they must occupy one or two display cells.

        texthl={group}
                Highlighting group used for the text item.


DELETING A SIGN                                       *:sign-undefine* *E155*

:sign undefine {name}
                Deletes a previously defined sign.  If signs with this {name}
                are still placed this will cause trouble.


LISTING SIGNS                                         *:sign-list* *E156*

:sign list      Lists all defined signs and their attributes.
```

```
:sign list {name}
                Lists one defined sign and its attributes.


PLACING SIGNS                                           *:sign-place* *E158*

:sign place {id} line={lnum} name={name} file={fname}
                Place sign defined as {name} at line {lnum} in file {fname}.
                                                        *:sign-fname*
                The file {fname} must already be loaded in a buffer.  The
                exact file name must be used, wildcards, $ENV and ~ are not
                expanded, white space must not be escaped.  Trailing white
                space is ignored.

                The sign is remembered under {id}, this can be used for
                further manipulation.  {id} must be a number.
                It's up to the user to make sure the {id} is used only once in
                each file (if it's used several times unplacing will also have
                to be done several times and making changes may not work as
                expected).

:sign place {id} line={lnum} name={name} buffer={nr}
                Same, but use buffer {nr}.

                                                        *E885*
:sign place {id} name={name} file={fname}
                Change the placed sign {id} in file {fname} to use the defined
                sign {name}.  See remark above about {fname} |:sign-fname|.
                This can be used to change the displayed sign without moving
                it (e.g., when the debugger has stopped at a breakpoint).

:sign place {id} name={name} buffer={nr}
                Same, but use buffer {nr}.


REMOVING SIGNS                                          *:sign-unplace* *E159*

:sign unplace {id} file={fname}
                Remove the previously placed sign {id} from file {fname}.
                See remark above about {fname} |:sign-fname|.

:sign unplace * file={fname}
                Remove all placed signs in file {fname}.

:sign unplace {id} buffer={nr}
                Remove the previously placed sign {id} from buffer {nr}.

:sign unplace * buffer={nr}
                Remove all placed signs in buffer {nr}.

:sign unplace {id}
                Remove the previously placed sign {id} from all files it
                appears in.

:sign unplace *
                Remove all placed signs.

:sign unplace
                Remove the placed sign at the cursor position.
```

```
LISTING PLACED SIGNS                                    *:sign-place-list*

:sign place file={fname}
               List signs placed in file {fname}.
               See remark above about {fname} |:sign-fname|.

:sign place buffer={nr}
               List signs placed in buffer {nr}.

:sign place    List placed signs in all files.


JUMPING TO A SIGN                                       *:sign-jump* *E157*

:sign jump {id} file={fname}
               Open the file {fname} or jump to the window that contains
               {fname} and position the cursor at sign {id}.
               See remark above about {fname} |:sign-fname|.
               If the file isn't displayed in window and the current file can
               not be |abandon|ed this fails.

:sign jump {id} buffer={nr}                                    *E934*
               Same, but use buffer {nr}.  This fails if buffer {nr} does not
               have a name.


 vim:tw=78:ts=8:ft=help:norl:
```