Get Specific Line From Variable Output

Problem:

I need to get a line from a commands output that is assigned to a bash variable.

```
#Gets the netstat values for the UPD packages

UDP_RAW_VALUE=`netstat -su`



##TRYING THIS BUT TOTALLY FAILED.

echo "$UDP_RAW_VALUE" | sed -n "$2"p
```

I just need to get a specific value from one of the command executions, which is

```
UDP_RAW_VALUE=`netstat -su`
```

The output of this specific command is like this:

```
-bash-4.2$ bash MyBashScript

IcmpMsg:

    InType0: 14464

    InType3: 12682

    InType8: 101

    InType11: 24

    OutType0: 101

    OutType3: 34385

    OutType8: 15840

Udp:

    931752 packets received

    889 packets to unknown port received.

    0 packet receive errors

    1007042 packets sent

    0 receive buffer errors

    0 send buffer errors
```

```
    IgnoredMulti: 647095
UdpLite:
IpExt:
    InMcastPkts: 1028470
    InBcastPkts: 552859
    InOctets: 233485843587
    OutOctets: 75548840236
    InMcastOctets: 44792084
    InBcastOctets: 167490770
    InNoECTPkts: 317265390
    InECT0Pkts: 25289
    InCEPkts: 686361
```

Simply, I need to read the **numeric** value of the **931752 packets received** result, *excluding the "packets received"* part. I have tried doing some regular expression. Then found this **sed** which looks promising but I have no idea how to use it, in terms of regular expression. I am pretty sure I need to complete this with regex.

I need the numerical value because I will compare a previous and current value and check if the difference is between the threshold. If not, send the mail.

ANSWER:

Just add this:

```
 PACKETS_RECEIVED=$(echo "${UDP_RAW_VALUE}" | sed -n 's/^ *\([0-9][0-9]*\) packets received$/\1/p')
```

It will print the content that you saved from the RAW output of the command. the `""` are important or it will print all the output in one line which is not what you want.

The sed command -n says do not print the lines by default. the `s///p` command tells is to substitute the pattern between the first two `/ /` with the `\1`, which is the group in the pattern that is between `\(` and `\)`; in our case that would be the number you want.

The `PACKET_RECEIVED=$()` part tells bash to run what is between the `$()` and assign the output to the variable.

perhaps another sed solution if you dont mind?

```
UDP_RAW_VALUE=`netstat -su|sed -n '/Udp/{n;s/[^0-9\n]//g;p}'`
```

`/Udp:/` #find all lines that have Udp

`n` # get the next line delete all all characters leaving you number

`p` #print

How To Read and Set Environmental and Shell Variables on a Linux VPS

PostedMarch 3, 2014 1.1m views Linux Basics Miscellaneous

Introduction

When interacting with your server through a shell session, there are many pieces of information that your shell compiles to determine its behavior and access to resources. Some of these settings are contained within configuration settings and others are determined by user input.

One way that the shell keeps track of all of these settings and details is through an area it maintains called the **environment**. The environment is an area that the shell builds every time that it starts a session that contains variables that define system properties.

In this guide, we will discuss how to interact with the environment and read or set environmental and shell variables interactively and through configuration files. We will be using an Ubuntu 12.04 VPS as an example, but these details should be relevant on any Linux system.

How the Environment and Environmental Variables Work

Every time a shell session spawns, a process takes place to gather and compile information that should be available to the shell process and its child processes. It obtains the data for these settings from a variety of different files and settings on the system.

Basically the environment provides a medium through which the shell process can get or set settings and, in turn, pass these on to its child processes.

The environment is implemented as strings that represent key-value pairs. If multiple values are passed, they are typically separated by colon (:) characters. Each pair will generally will look something like this:

```
KEY=value1:value2:...
```

If the value contains significant white-space, quotations are used:

```
KEY="value with spaces"
```

The keys in these scenarios are variables. They can be one of two types, environmental variables or shell variables.

Environmental variables are variables that are defined for the current shell and are inherited by any child shells or processes. Environmental variables are used to pass information into processes that are spawned from the shell.

Shell variables are variables that are contained exclusively within the shell in which they were set or defined. They are often used to keep track of ephemeral data, like the current working directory.

By convention, these types of variables are usually defined using all capital letters. This helps users distinguish environmental variables within other contexts.

Printing Shell and Environmental Variables

Each shell session keeps track of its own shell and environmental variables. We can access these in a few different ways.

We can see a list of all of our environmental variables by using the `env` or `printenv` commands. In their default state, they should function exactly the same:

```
printenv
```

```
SHELL=/bin/bash
TERM=xterm
USER=demouser
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;33;01:cd=40;3
3;01:or=40;31;01:su=37;41:sg=30;43:ca:...
MAIL=/var/mail/demouser
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
PWD=/home/demouser
LANG=en_US.UTF-8
```

```
SHLVL=1
HOME=/home/demouser
LOGNAME=demouser
LESSOPEN=| /usr/bin/lesspipe %s
LESSCLOSE=/usr/bin/lesspipe %s %s
_=/usr/bin/printenv
```

This is fairly typical of the output of both `printenv` and `env`. The difference between the two commands is only apparent in their more specific functionality. For instance, with `printenv`, you can requests the values of individual variables:

```
printenv SHELL
```

```
/bin/bash
```

On the other hand, `env` let's you modify the environment that programs run in by passing a set of variable definitions into a command like this:

```
env VAR1="blahblah" command_to_run command_options
```

Since, as we learned above, child processes typically inherit the environmental variables of the parent process, this gives you the opportunity to override values or add additional variables for the child.

As you can see from the output of our `printenv` command, there are quite a few environmental variables set up through our system files and processes without our input.

These show the environmental variables, but how do we see shell variables?

The `set` command can be used for this. If we type `set` without any additional parameters, we will get a list of all shell variables, environmental variables, local variables, and shell functions:

```
set
```

```
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extglob:extquote:force_fignore:histappen
d:interactive_comments:login_shell:progcomp:promptvars:sourcepath
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
. . .
```

This is usually a huge list. You probably want to pipe it into a pager program to deal with the amount of output easily:

```
set | less
```

The amount of additional information that we receive back is a bit overwhelming. We probably do not need to know all of the bash functions that are defined, for instance.

We can clean up the output by specifying that `set` should operate in POSIX mode, which won't print the shell functions. We can execute this in a sub-shell so that it does not change our current environment:

```
(set -o posix; set)
```

This will list all of the environmental and shell variables that are defined.

We can attempt to compare this output with the output of the `env` or `printenv` commands to try to get a list of only shell variables, but this will be imperfect due to the different ways that these commands output information:

```
comm -23 <(set -o posix; set | sort) <(env | sort)
```

This will likely still include a few environmental variables, due to the fact that the `set` command outputs quoted values, while the `printenv` and `env` commands do not quote the values of strings.

This should still give you a good idea of the environmental and shell variables that are set in your session.

These variables are used for all sorts of things. They provide an alternative way of setting persistent values for the session between processes, without writing changes to a file.

Common Environmental and Shell Variables

Some environmental and shell variables are very useful and are referenced fairly often.

Here are some common environmental variables that you will come across:

- **SHELL**: This describes the shell that will be interpreting any commands you type in. In most cases, this will be bash by default, but other values can be set if you prefer other options.
- **TERM**: This specifies the type of terminal to emulate when running the shell. Different hardware terminals can be emulated for different operating requirements. You usually won't need to worry about this though.
- **USER**: The current logged in user.
- **PWD**: The current working directory.
- **OLDPWD**: The previous working directory. This is kept by the shell in order to switch back to your previous directory by running `cd -`.

- **LS_COLORS**: This defines color codes that are used to optionally add colored output to the `ls` command. This is used to distinguish different file types and provide more info to the user at a glance.
- **MAIL**: The path to the current user's mailbox.
- **PATH**: A list of directories that the system will check when looking for commands. When a user types in a command, the system will check directories in this order for the executable.
- **LANG**: The current language and localization settings, including character encoding.
- **HOME**: The current user's home directory.
- _: The most recent previously executed command.
  In addition to these environmental variables, some shell variables that you'll often see are:

- **BASHOPTS**: The list of options that were used when bash was executed. This can be useful for finding out if the shell environment will operate in the way you want it to.
- **BASH_VERSION**: The version of bash being executed, in human-readable form.
- **BASH_VERSINFO**: The version of bash, in machine-readable output.
- **COLUMNS**: The number of columns wide that are being used to draw output on the screen.
- **DIRSTACK**: The stack of directories that are available with the `pushd` and `popd` commands.
- **HISTFILESIZE**: Number of lines of command history stored to a file.
- **HISTSIZE**: Number of lines of command history allowed in memory.
- **HOSTNAME**: The hostname of the computer at this time.
- **IFS**: The internal field separator to separate input on the command line. By default, this is a space.
- **PS1**: The primary command prompt definition. This is used to define what your prompt looks like when you start a shell session. The `PS2` is used to declare secondary prompts for when a command spans multiple lines.
- **SHELLOPTS**: Shell options that can be set with the `set` option.
- **UID**: The UID of the current user.

Setting Shell and Environmental Variables

To better understand the difference between shell and environmental variables, and to introduce the syntax for setting these variables, we will do a small demonstration.

Creating Shell Variables

We will begin by defining a shell variable within our current session. This is easy to accomplish; we only need to specify a name and a value. We'll adhere to the convention of keeping all caps for the variable name, and set it to a simple string.

```
TEST_VAR='Hello World!'
```

Here, we've used quotations since the value of our variable contains a space. Furthermore, we've used single quotes because the exclamation point is a special character in the bash shell that normally expands to the bash history if it is not escaped or put into single quotes.

We now have a shell variable. This variable is available in our current session, but will not be passed down to child processes.

We can see this by grepping for our new variable within the `set` output:

```
set | grep TEST_VAR
```

```
TEST_VAR='Hello World!'
```

We can verify that this is not an environmental variable by trying the same thing with `printenv`:

```
printenv | grep TEST_VAR
```

No out should be returned.

Let's take this as an opportunity to demonstrate a way of accessing the value of any shell or environmental variable.

```
echo $TEST_VAR
```

```
Hello World!
```

As you can see, reference the value of a variable by preceding it with a `$` sign. The shell takes this to mean that it should substitute the value of the variable when it comes across this.

So now we have a shell variable. It shouldn't be passed on to any child processes. We can spawn a *new* bash shell from within our current one to demonstrate:

```
bash
echo $TEST_VAR
```

If we type `bash` to spawn a child shell, and then try to access the contents of the variable, nothing will be returned. This is what we expected.

Get back to our original shell by typing `exit`:

```
exit
```

Creating Environmental Variables

Now, let's turn our shell variable into an environmental variable. We can do this by *exporting* the variable. The command to do so is appropriately named:

```
export TEST_VAR
```
This will change our variable into an environmental variable. We can check this by checking our environmental listing again:

```
printenv | grep TEST_VAR
```

```
TEST_VAR=Hello World!
```
This time, our variable shows up. Let's try our experiment with our child shell again:

```
bash
echo $TEST_VAR
```

```
Hello World!
```
Great! Our child shell has received the variable set by its parent. Before we exit this child shell, let's try to export another variable. We can set environmental variables in a single step like this:

```
export NEW_VAR="Testing export"
```
Test that it's exported as an environmental variable:

```
printenv | grep NEW_VAR
```

```
NEW_VAR=Testing export
```
Now, let's exit back into our original shell:

```
exit
```
Let's see if our new variable is available:

```
echo $NEW_VAR
```
Nothing is returned.

This is because environmental variables are only passed to child processes. There isn't a built-in way of setting environmental variables of the parent shell. This is good in most cases and prevents programs from affecting the operating environment from which they were called.

The `NEW_VAR` variable was set as an environmental variable in our child shell. This variable would be available to itself and any of **its** child shells and processes. When we exited back into our main shell, that environment was destroyed.

Demoting and Unsetting Variables

We still have our `TEST_VAR` variable defined as an environmental variable. We can change it back into a shell variable by typing:

```
export -n TEST_VAR
```
It is no longer an environmental variable:

```
printenv | grep TEST_VAR
```
However, it is still a shell variable:

```
set | grep TEST_VAR
```

```
TEST_VAR='Hello World!'
```
If we want to completely unset a variable, either shell or environmental, we can do so with the `unset` command:

```
unset TEST_VAR
```
We can verify that it is no longer set:

```
echo $TEST_VAR
```
Nothing is returned because the variable has been unset.

Setting Environmental Variables at Login

We've already mentioned that many programs use environmental variables to decide the specifics of how to operate. We do not want to have to set important variables up every time we start a new shell session, and we have already seen how many variables are already set upon login, so how do we make and define variables automatically?

This is actually a more complex problem than it initially seems, due to the numerous configuration files that the bash shell reads depending on how it is started.

The Difference between Login, Non-Login, Interactive, and Non-Interactive Shell Sessions

---

The bash shell reads different configuration files depending on how the session is started.

One distinction between different sessions is whether the shell is being spawned as a "login" or "non-login" session.

A **login** shell is a shell session that begins by authenticating the user. If you are signing into a terminal session or through SSH and authenticate, your shell session will be set as a "login" shell.

If you start a new shell session from within your authenticated session, like we did by calling the `bash` command from the terminal, a **non-login** shell session is started. You were were not asked for your authentication details when you started your child shell.

Another distinction that can be made is whether a shell session is interactive, or non-interactive.

An **interactive** shell session is a shell session that is attached to a terminal. A **non-interactive** shell session is one is not attached to a terminal session.

So each shell session is classified as either login or non-login and interactive or non-interactive.

A normal session that begins with SSH is usually an interactive login shell. A script run from the command line is usually run in a non-interactive, non-login shell. A terminal session can be any combination of these two properties.

Whether a shell session is classified as a login or non-login shell has implications on which files are read to initialize the shell session.

A session started as a login session will read configuration details from the `/etc/profile` file first. It will then look for the first login shell configuration file in the user's home directory to get user-specific configuration details.

It reads the first file that it can find out of `~/.bash_profile`, `~/.bash_login`, and `~/.profile` and does not read any further files.

In contrast, a session defined as a non-login shell will read `/etc/bash.bashrc` and then the user-specific `~/.bashrc` file to build its environment.

Non-interactive shells read the environmental variable called `BASH_ENV` and read the file specified to define the new environment.

Implementing Environmental Variables

As you can see, there are a variety of different files that we would usually need to look at for placing our settings.

This provides a lot of flexibility that can help in specific situations where we want certain settings in a login shell, and other settings in a non-login shell. However, most of the time we will want the same settings in both situations.

Fortunately, most Linux distributions configure the login configuration files to source the non-login configuration files. This means that you can define environmental variables that you want in both inside the non-login configuration files. They will then be read in both scenarios.

We will usually be setting user-specific environmental variables, and we usually will want our settings to be available in both login and non-login shells. This means that the place to define these variables is in the `~/.bashrc` file.

Open this file now:

```
nano ~/.bashrc
```

This will most likely contain quite a bit of data already. Most of the definitions here are for setting bash options, which are unrelated to environmental variables. You can set environmental variables just like you would from the command line:

```
export VARNAME=value
```

We can then save and close the file. The next time you start a shell session, your environmental variable declaration will be read and passed on to the shell environment. You can force your current session to read the file now by typing:

```
source ~/.bashrc
```

If you need to set system-wide variables, you may want to think about adding them to `/etc/profile`, `/etc/bash.bashrc`, or `/etc/environment`.

Conclusion

Environmental and shell variables are always present in your shell sessions and can be very useful. They are an interesting way for a parent process to set configuration details for its children, and are a way of setting options outside of files.

This has many advantages in specific situations. For instance, some deployment mechanisms rely on environmental variables to configure authentication information. This is useful because it does not require keeping these in files that may be seen by outside parties.

There are plenty of other, more mundane, but more common scenarios where you will need to read or alter the environment of your system. These tools and techniques should give you a good foundation for making these changes and using them correctly.

sed(1) - Linux man page

Name

sed - stream editor for filtering and transforming text

Synopsis

sed [*OPTION*]... *{script-only-if-no-other-script}* [*input-file*]...

Description

Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as *ed*), *sed* works by making only one pass over the **input**(s), and is consequently more efficient. But it is *sed*'s ability to filter text in a pipeline which particularly distinguishes it from other types of editors.

-n, **--quiet**, **--silent**

suppress automatic printing of pattern space

-e script, **--expression**=*script*

add the script to the commands to be executed

-f script-file, **--file**=*script-file*

add the contents of script-file to the commands to be executed

--follow-symlinks

follow symlinks when processing in place; hard links will still be broken.

-i[SUFFIX], **--in-place**[=*SUFFIX*]

edit files in place (makes backup if extension supplied). The default operation mode is to break symbolic and hard links. This can be changed with **--follow-symlinks** and **--copy**.

-c, **--copy**

use copy instead of rename when shuffling files in **-i** mode. While this will avoid breaking links (symbolic or hard), the resulting editing operation is not atomic. This is rarely the desired mode; **--follow-symlinks** is usually enough, and it is both faster and more secure.

-l N, **--line-length**=*N*

specify the desired line-wrap length for the 'l' command

--posix

disable all GNU extensions.

-r, **--regexp-extended**

use extended regular expressions in the script.

-s, **--separate**

consider files as separate rather than as a single continuous long stream.

-u, **--unbuffered**

load minimal amounts of data from the input files and flush the output buffers more often

--help

display this help and exit

--version

output version information and exit

If no **-e**, **--expression**, **-f**, or **--file** option is given, then the first non-option argument is taken as the sed script to interpret. All remaining arguments are names of input files; if no input files are specified, then the standard input is read.

GNU sed home page: <http://www.gnu.org/software/sed/>. General help using GNU software: <http://www.gnu.org/gethelp/>. E-mail bug reports to:< bug-gnu-utils@gnu.org>. Be sure to include the word ''sed'' somewhere in the ''Subject:'' field.

Command Synopsis

This is just a brief synopsis of *sed* commands to serve as a reminder to those who already know *sed*; other documentation (such as the texinfo document) must be consulted for fuller descriptions.

Zero-address ''commands''

: *label*

Label for **b** and **t** commands.

#comment

The comment extends until the next newline (or the end of a **-e** script fragment).

}

The closing bracket of a { } block.

Zero- or One- address commands

=

Print the current line number.

a \

text

Append *text*, which has each embedded newline preceded by a backslash.

i \

text

Insert *text*, which has each embedded newline preceded by a backslash.

q [*exit-code*]

Immediately quit the *sed* script without processing any more input, except that if auto-print is not disabled the current pattern space will be printed. The exit code argument is a GNU extension.

Q [*exit-code*]

Immediately quit the *sed* script without processing any more input. This is a GNU extension.

r *filename*

Append text read from *filename*.

R *filename*

Append a line read from *filename*. Each invocation of the command reads a line from the file. This is a GNU extension.

Commands which accept address ranges

{

Begin a block of commands (end with a }).

b *label*

Branch to *label*; if *label* is omitted, branch to end of script.

t *label*

If a s/// has done a successful substitution since the last input line was read and since the last t or T command, then branch to *label*; if *label* is omitted, branch to end of script.

T *label*

If no s/// has done a successful substitution since the last input line was read and since the last t or T command, then branch to *label*; if *label* is omitted, branch to end of script. This is a GNU extension.

c \

text

Replace the selected lines with *text*, which has each embedded newline preceded by a backslash.

d

Delete pattern space. Start next cycle.

D

Delete up to the first embedded newline in the pattern space. Start next cycle, but skip reading from the input if there is still data in the pattern space.

h H

Copy/append pattern space to hold space.

g G

Copy/append hold space to pattern space.

x

Exchange the contents of the hold and pattern spaces.

l

List out the current line in a ''visually unambiguous'' form.

l *width*

List out the current line in a ''visually unambiguous'' form, breaking it at *width* characters. This is a GNU extension.

n N

Read/append the next line of input into the pattern space.

p

Print the current pattern space.

P

Print up to the first embedded newline of the current pattern space.

s/*regexp*/*replacement*/

Attempt to match *regexp* against the pattern space. If successful, replace that portion matched with *replacement*. The *replacement* may contain the special character **&** to refer to that portion of the pattern space which matched, and the special escapes \1 through \9 to refer to the corresponding matching sub-expressions in the *regexp*.

w *filename*

Write the current pattern space to *filename*.

W *filename*

Write the first line of the current pattern space to *filename*. This is a GNU extension.

y/*source*/*dest*/

Transliterate the characters in the pattern space which appear in *source* to the corresponding character in *dest*.

Addresses

Sed commands can be given with no addresses, in which case the command will be executed for all input lines; with one address, in which case the command will only be executed for input lines which match that address; or with two addresses, in which case the command will be executed for all input lines which match the inclusive range of lines starting from the first address and continuing to the second address. Three things to note about address ranges: the syntax is *addr1*,*addr2* (i.e., the addresses are separated by a comma); the line which *addr1* matched will always be accepted, even if *addr2* selects an earlier line; and if *addr2* is a *regexp*, it will not be tested against the line that *addr1* matched.

After the address (or address-range), and before the command, a **!** may be inserted, which specifies that the command shall only be executed if the address (or address-range) does **not** match.

The following address types are supported:

number

Match only the specified line *number*.

first~step

> Match every *step*'th line starting with line *first*. For example, ''sed -n 1~2p'' will print all the odd-numbered lines in the input stream, and the address 2~5 will match every fifth line, starting with the second. *first* can be zero; in this case, *sed* operates as if it were equal to *step*. (This is an extension.)

$

Match the last line.

/regexp/

> Match lines matching the regular expression *regexp*.

c*regexp***c**

> Match lines matching the regular expression *regexp*. The **c** may be any character.

GNU *sed* also supports some special 2-address forms:

0,*addr2*

> Start out in "matched first address" state, until *addr2* is found. This is similar to 1,*addr2*, except that if *addr2* matches the very first line of input the 0,*addr2* form will be at the end of its range, whereas the 1,*addr2* form will still be at the beginning of its range. This works only when *addr2* is a regular expression.

addr1,+*N*

> Will match *addr1* and the *N* lines following *addr1*.

addr1,~*N*

> Will match *addr1* and the lines following *addr1* until the next line whose input line number is a multiple of *N*.

Regular Expressions

POSIX.2 BREs *should* be supported, but they aren't completely because of performance problems. The **\n** sequence in a regular expression matches the newline character, and similarly for **\a**, **\t**, and other sequences.

Bugs

E-mail bug reports to **bonzini@gnu.org**. Be sure to include the word ''sed'' somewhere in the ''Subject:'' field. Also, please include the output of ''sed --version'' in the body of your report if at all possible.

Copyright

GNU sed home page: <http://www.gnu.org/software/sed/>. General help using GNU software: <http://www.gnu.org/gethelp/>. E-mail bug reports to:< bug-gnu-utils@gnu.org>. Be sure to include the word ''sed'' somewhere in the ''Subject:'' field.

See Also

awk(1), **ed**(1), **grep**(1), **tr**(1), **perlre**(1), sed.info, any of various books on *sed*, the *sed* FAQ (http://sed.sf.net/grabbag/tutorials/sedfaq.txt), http://sed.sf.net/grabbag/.

The full documentation for **sed** is maintained as a Texinfo manual. If the **info** and **sed** programs are properly installed at your site, the command

info sed

should give you access to the complete manual.

Referenced By

bbe(1), **cpuset**(7), **dialrules**(5), **fetchlog**(1), **flowdumper**(1), **formail**(1), **iostat2pcp**(1), **ksh**(1), **libarchive-formats**(5), **med**(1), **mk-configure**(7), **mksh**(1), **nawk**(1), **nc**(1), **pagermap**(5), **rpl**(1), **rubibtex**(1), **rumakeindex**(1), **virt-edit**(1), **zipinfo**(1)

cut(1) - Linux man page

Name

cut - remove sections from each line of files

Synopsis

cut *OPTION*... [*FILE*]...

Description

Print selected parts of lines from each FILE to standard output.

Mandatory arguments to long options are mandatory for short options too.

-b, **--bytes**=*LIST*

select only these bytes

-c, **--characters**=*LIST*

select only these characters

-d, **--delimiter**=*DELIM*

use DELIM instead of TAB for field delimiter

-f, **--fields**=*LIST*

select only these fields; also print any line that contains no delimiter character, unless the **-s** option is specified

-n

with **-b**: don't split multibyte characters

--complement

complement the set of selected bytes, characters or fields

-s, **--only-delimited**

do not print lines not containing delimiters

--output-delimiter=*STRING*

use STRING as the output delimiter the default is to use the input delimiter

--help

display this help and exit

--version

output version information and exit

Use one, and only one of **-b**, **-c** or **-f**. Each LIST is made up of one range, or many ranges separated by commas. Selected input is written in the same order that it is read, and is written exactly once. Each range is one of:

N

N'th byte, character or field, counted from 1

N-

from N'th byte, character or field, to end of line

N-M

from N'th to M'th (included) byte, character or field

-M

from first to M'th (included) byte, character or field

With no FILE, or when FILE is -, read standard input.

Author

Written by David M. Ihnat, David MacKenzie, and Jim Meyering.

Reporting Bugs

Report cut bugs to bug-coreutils@gnu.org
GNU coreutils home page: <http://www.gnu.org/software/coreutils/>
General help using GNU software: <http://www.gnu.org/gethelp/>
Report cut translation bugs to <http://translationproject.org/team/>

Copyright

See Also

The full documentation for **cut** is maintained as a Texinfo manual. If the **info** and **cut** programs are properly installed at your site, the command

info coreutils aqcut invocationaq

should give you access to the complete manual.

Referenced By

dbview(1), **file**(1), **ksh93**(1)