Defining a variable with or without export

Asked 15 years, 5 months ago Modified 4 months ago Viewed 526k times

1256

What is `export` for?

What is the difference between:

```
export name=value
```

and

```
name=value
```

linux bash shell

Share

Improve this question

Follow

edited Nov 5, 2013 at 16:15

Will Vousden

33.3k ● 9 ● 87 ● 97

asked Jul 21, 2009 at 9:09

flybywire

273k ● 200 ● 404 ● 509

2 See also: stackoverflow.com/questions/68529921/… – Roland Jul 26, 2021 at 13:45

16 Answers

Sorted by: Highest score (default)

1371

`export` makes the variable available to sub-processes.

That is,

```
export name=value
```

means that the variable name is available to *any process* you run from that shell process. If you want a process to make use of this variable, use `export`, and run the process from that shell.

```
name=value
```

means the variable scope is restricted to the shell, and is not available to any other process. You would use this for (say) loop variables, temporary variables etc.

It's important to note that exporting a variable doesn't make it available to parent processes. That is, specifying and exporting a variable in a spawned process doesn't make it available in the process that launched it.

Share

Improve this answer

Follow

edited Oct 1, 2013 at 10:40

answered Jul 21, 2009 at 9:45

Brian Agnew

272k ● 38 ● 339 ● 442

132 Specifically export makes the variable available to child processes via the environment. – Beano Jul 21, 2009 at 13:35

24 I'd also add that if the export is in a file that you "source" (like . filename) then it exports it to your working environment as well. – rogerdpack Sep 17, 2013 at 22:44

10 @rogerdpack can't you do that without export? cat > blah \n a=hi \n . blah; echo $a; outputs 'hi' for me. – David Winiecki Sep 30, 2013 at 23:49

4 Nice it does work even without the export. So I guess when sourcing a file, if you use export it will be reflected in child processes, if you don't it will just affect the local bash environment... – rogerdpack Oct 1, 2013 at 15:01

38 There's one edge-case to this; `name=value command` *does* make the variable available in the sub-process `command`. – Oliver Charlesworth Dec 23, 2015 at 11:09

344

To illustrate what the other answers are saying:

```
$ export foo="Hello, World"
$ bar="Goodbye"

$ echo $foo
Hello, World
$ echo $bar
Goodbye

$ bash
bash-3.2$ echo $foo
Hello, World
bash-3.2$ echo $bar

bash-3.2$
```

Share

Improve this answer

Follow

edited May 1 at 0:55

user23375353

answered Jul 21, 2009 at 9:56

alxp

6,281 ● 1 ● 24 ● 19

13 One more example for this `al$ foobar="Whatever" bash` – Alun Jun 15, 2016 at 20:57

1 Just in case someone wants to try this in Bash with arrays (like I did...) then a heads-up: it can't be done. – toraritte May 10, 2021 at 19:43

98

It has been said that it's not necessary to export in bash when spawning subshells, while others said the exact opposite. It is important to note the difference between subshells (those that are created by `()`, `` `` ``, `$()` or loops) and subprocesses (processes that are invoked by name, for example a literal `bash` appearing in your script).

- Sub*shells* **will** have access to all variables from the parent, regardless of their exported state.
- Sub*processes* will **only** see the exported variables.

What is common in these two constructs is that neither can pass variables back to the parent shell.

```
$ noexport=noexport; export export=export; (echo subshell: $noexport $export;
subshell=subshell); bash -c 'echo subprocess: $noexport $export;
subprocess=subprocess'; echo parent: $subshell $subprocess
subshell: noexport export
subprocess: export
parent:
```

There is one more source of confusion: some think that 'forked' subprocesses are the ones that don't see non-exported variables. Usually fork()s are immediately followed by exec()s, and that's why it would seem that the fork() is the thing to look for, while in fact it's the exec(). You can run commands without fork()ing first with the `exec` command, and processes started by this method will also have no access to unexported variables:

```
$ noexport=noexport; export export=export; exec bash -c 'echo execd process:
$noexport $export; execd=execd'; echo parent: $execd
execd process: export
```

Note that we don't see the `parent:` line this time, because we have replaced the parent shell with the `exec` command, so there's nothing left to execute that command.

Share

Improve this answer

Follow

edited Nov 27, 2018 at 22:57

lazarusleroux

320 ● 2 ● 14

answered Feb 12, 2014 at 8:16

Matyas Koszik

1,196 ● 7 ● 8

I've never seen a loop that (by itself) created a subshell; OTOH a pipeline does (always for pieces other than the last, sometimes for the last depending on your shell, version, and

options). Backgrounding (`&`) also creates a subshell. – dave_thompson_085 Sep 18, 2017 at 23:38

2 What about these `var=asdf bash -c 'echo $var'` or `var=asdf exec bash -c 'echo $var'`? The output is `asdf`. The `;` makes difference if placed after the variable definition. What would be the explanation? It looks like the `var` (with no `;`) regards to spawned subprocess somehow, due to origin shell has nothing to do with it. `echo $var` prints nothing if executed on second line. But one lined `var=asdf bash -c 'echo $var'; echo $var` gives `asdf\nasdf`. – 4xy Jul 13, 2019 at 13:47

1 @4xy That's a different case altogether; `var=value command` sets the variable `var` to the value `value` for the duration of the command `command`, in that command's environment. This is vaguely similar to what the `env` command does. – tripleee Apr 19, 2021 at 4:54

87

This answer is *wrong* but retained for historical purposes. See 2nd edit below.

Others have answered that export makes the variable available to subshells, and that is correct but merely a side effect. When you export a variable, it puts that variable in the environment of the current shell (ie the shell calls `putenv(3)` or `setenv(3)`). The environment of a process is inherited across exec, making the variable visible in subshells.

Edit (with 5 years' perspective): This is a silly answer. The purpose of 'export' is to make variables "be in the environment of subsequently executed commands", whether those commands be subshells or subprocesses. A naive implementation would be to simply put the variable in the environment of the shell, but this would make it impossible to implement `export -p`.

2nd Edit (with another 5 years in passing). This answer is just bizarre. Perhaps I had some reason at one point to claim that bash puts the exported variable into its own environment, but those reasons were not given here and are now lost to history. See Exporting a function local variable to the environment.

Share

Improve this answer

Follow

edited May 8, 2023 at 17:21

Karl Wilbur

6,177 ● 3 ● 46 ● 57

answered Jul 21, 2009 at 10:33

William Pursell

212k ● 48 ● 274 ● 313

8 Note that this is not entirely true. In `bash`, export does indeed add the variable to the environment of the current shell, but this is not the case with `dash`. It seems to me that adding the variable to the environment of the current shell is the simplest way to implement the semantics of `export`, but that behavior is not mandated. – William Pursell Jul 2, 2013 at 13:09

8 I'm not sure what `dash` has to do with this. The original poster was asking specifically about `bash`. – Starfish Sep 7, 2013 at 4:17

18 The question is tagged `bash` but applies equally to any bourne-shell variant. Being overly specific and providing answers that apply only to `bash` is a great evil. – William Pursell Sep 7, 2013 at 7:17

15 `bash` is the jQuery of the shell. – Potherca May 24, 2014 at 17:10

3 `export makes the variable available to subshells, and that is correct` This is a very confusing use of terminology. Subshells don't need `export` to inherit variables. Subprocesses do. – Amit Naidu May 19, 2018 at 18:37

39

`export NAME=value` for settings and variables that have meaning to a subprocess.

`NAME=value` for temporary or loop variables private to the current shell process.

In more detail, `export` marks the variable name in the environment that copies to a subprocesses and their subprocesses upon creation. No name or value is ever copied back from the subprocess.

- A common error is to place a space around the equal sign:

```
$ export FOO = "bar"
bash: export: `=': not a valid identifier
```

- Only the exported variable (`B`) is seen by the subprocess:

```
$ A="Alice"; export B="Bob"; echo "echo A is \$A. B is \$B" | bash
A is . B is Bob
```

- Changes in the subprocess do not change the main shell:

```
$ export B="Bob"; echo 'B="Banana"' | bash; echo $B
Bob
```

- Variables marked for export have values copied when the subprocess is created:

```
$ export B="Bob"; echo '(sleep 30; echo "Subprocess 1 has B=$B")' | bash &
[1] 3306
$ B="Banana"; echo '(sleep 30; echo "Subprocess 2 has B=$B")' | bash
Subprocess 1 has B=Bob
Subprocess 2 has B=Banana
[1]+  Done         echo '(sleep 30; echo "Subprocess 1 has B=$B")' | bash
```

- Only exported variables become part of the environment (`man environ`):

```
$ ALICE="Alice"; export BOB="Bob"; env | grep "ALICE\|BOB"
BOB=Bob
```

So, now it should be as clear as is the summer's sun! Thanks to Brain Agnew, alexp, and William Prusell.

Share

Improve this answer

Follow

edited Apr 24, 2013 at 18:19

answered Apr 24, 2013 at 17:10

Charles Merriam

20.5k ● 7 ● 75 ● 84

22

It should be noted that you can export a variable and later change the value. The variable's changed value will be available to child processes. Once export has been set for a variable you must do `export -n <var>` to remove the property.

```
$ K=1
$ export K
$ K=2
$ bash -c 'echo ${K-unset}'
2
$ export -n K
$ bash -c 'echo ${K-unset}'
unset
```

Share

Improve this answer

Follow

edited Feb 12, 2017 at 14:16

Mateusz Piotrowski

9,087 ● 11 ● 59 ● 82

answered Feb 10, 2016 at 16:36

Brian S. Wilson

629 ● 6 ● 11

5 Thanks, this is exactly the information I was looking for because I saw a script which used environment variables and then "re-exported" them w/ a new value, and I was wondering if it was necessary. – Mike Lippert Jun 1, 2016 at 16:28

18

`export` will make the variable available to all shells forked from the current shell.

Share Improve this answer Follow

answered Jul 21, 2009 at 9:10

user36457

Hi John, do you know whether the position of this export matters? Should I put it at the bottom of the makefile or anywhere is okay? – Guodong Hu Aug 12, 2020 at 3:08

@leolehu `export` in a `Makefile` is generally a separate question. Quick experimentation reveals that, predictably, the location of the `export` is unimportant. (Predictable because `make` will parse and process the entire `Makefile` and *then* attempt to resolve dependencies to figure out which target(s) to run.) – tripleee Jun 5, 2022 at 10:59

16

As you might already know, UNIX allows processes to have a set of environment variables, which are key/value pairs, both key and value being strings. Operating system is responsible for keeping these pairs for each process separately.

Program can access its environment variables through this UNIX API:

- `char *getenv(const char *name);`
- `int setenv(const char *name, const char *value, int override);`
- `int unsetenv(const char *name);`

Processes also inherit environment variables from parent processes. Operating system is responsible for creating a copy of all "envars" at the moment the child process is created.

Bash, among other shells, is capable of setting its environment variables on user request. This is what `export` exists for.

`export` is a Bash command to set environment variable for Bash. All variables set with this command would be inherited by all processes that this Bash would create.

More on Environment in Bash

Another kind of variable in Bash is internal variable. Since Bash is not just interactive shell, it is in fact a script interpreter, as any other interpreter (e.g. Python) it is capable of keeping its own set of variables. It should be mentioned that Bash (unlike Python) supports only string variables.

Notation for defining Bash variables is `name=value`. These variables stay inside Bash and have nothing to do with environment variables kept by operating system.

More on Shell Parameters (including variables)

Also worth noting that, according to Bash reference manual:

> The environment for any simple command or function may be augmented temporarily by prefixing it with parameter assignments, as described in Shell Parameters. These assignment statements affect only the environment seen by that command.

To sum things up:

- `export` is used to set environment variable in operating system. This variable will be available to all child processes created by current Bash process ever after.

- Bash variable notation (name=value) is used to set local variables available only to current process of bash
- Bash variable notation prefixing another command creates environment variable only for scope of that command.

Share Improve this answer Follow

answered Jun 19, 2015 at 16:48

progalgo
358 ● 3 ● 7

1 bash vars don't support as many types as Python, but do have string, integer, and two kinds of array ('indexed'/traditional and 'associative' which is similar to awk array, perl hash, or Python dict). Other shells vary; only string is *portable*. – dave_thompson_085 Sep 18, 2017 at 23:58

@dave_thompson_085 - Actually, all are stored as string arrays and automatically converted if necessary for arithmetic, etc. A common variable like `A="string"` is actually identical to `A[0]="string"`. In fact, after saying `A="string"`, you could concatenate more strings to the 1-string array with `A[1]="string2"`, `A+=(string3 string4 "string 5 is longer")` then `echo "${A[@]}"` to print them. Note that it would require something like feeding the array to a `printf` command in order to get some kind of delimiter between strings since the default is a space and string5 contains spaces. – DocSalvager Jul 22, 2020 at 5:39

@DocSalvager: `export a b; a=this; b[0]=that; env | grep ^[ab]` isn't identical. In C/C++/Java `float` and `double` are interchangeable in *some* situations but they are still different types. – dave_thompson_085 Jul 28, 2020 at 0:31

9

The accepted answer implies this, but I'd like to make explicit the connection to shell builtins:

As mentioned already, `export` will make a variable available to both the shell and children. If `export` is *not* used, the variable will only be available in the shell, and only shell *builtins* can access it.

That is,

```
tango=3
env | grep tango # prints nothing, since env is a child process
set | grep tango # prints tango=3 - "type set" shows `set` is a shell builtin
```

Share

Improve this answer

Follow

edited Nov 15, 2017 at 23:43

answered Sep 22, 2017 at 7:33

flow2k
4,317 ● 49 ● 59

Two of the creators of UNIX, Brian Kernighan and Rob Pike, explain this in their book "The UNIX Programming Environment". Google for the title and you'll easily find a pdf version.

They address shell variables in section 3.6, and focus on the use of the `export` command at the end of that section:

> When you want to make the value of a variable accessible in sub-shells, the shell's export command should be used. (You might think about why there is no way to export the value of a variable from a sub-shell to its parent).

Share Improve this answer Follow

answered Jul 11, 2016 at 17:50

Dan Carter
587 ● 5 ● 8

Here's yet another example:

```
VARTEST="value of VARTEST"
#export VARTEST="value of VARTEST"
sudo env | grep -i vartest
sudo echo ${SUDO_USER} ${SUDO_UID}:${SUDO_GID} "${VARTEST}"
sudo bash -c 'echo ${SUDO_USER} ${SUDO_UID}:${SUDO_GID} "${VARTEST}"'
```

Only by using export VARTEST the value of VARTEST is available in sudo bash -c '...'!

For further examples see:

- http://mywiki.wooledge.org/SubShell
- bash-hackers.org/wiki/doku.php/scripting/processtree

Share Improve this answer Follow

answered Jul 22, 2009 at 7:06

soxie

Just to show the difference between an exported variable being in the environment (env) and a non-exported variable not being in the environment:

If I do this:

```
$ MYNAME=Fred
$ export OURNAME=Jim
```

then only $OURNAME appears in the env. The variable $MYNAME is not in the env.

```
$ env | grep NAME
OURNAME=Jim
```

but the variable $MYNAME does exist in the shell

```
$ echo $MYNAME
Fred
```

Share Improve this answer Follow

answered Feb 5, 2014 at 11:00

Will

1,739 ● 2 ● 14 ● 18

Hi Will, can I export a variable before the variable declaration? like export OURNAME and then OURNAME=Jim? – Guodong Hu Aug 12, 2020 at 3:10

@leoleohu if you export OURNAME before you assign it, you'll just export an empty string. – ingernet Oct 30, 2020 at 16:51

@ingernet even if you export variable before assigning it, any value assigned to that variable until before the child process is invoked will be seen by the child process. But once the child process is invoked, any updates to the exported variable done in the parent process will not be seen by the child process, that's because the variables are copied by value during process 'exec' call – Pavan May 19, 2021 at 18:19

3

By default, variables created within a script are only available to the current shell; child processes (sub-shells) will not have access to values that have been set or modified. Allowing child processes to see the values, requires use of the export command.

Share Improve this answer Follow

answered Dec 9, 2014 at 21:39

Amjad

3,638 ● 2 ● 22 ● 19

1

As yet another corollary to the existing answers here, let's rephrase the problem statement.

The answer to "should I `export`" is identical to the answer to the question "Does your subsequent code run a command which *implicitly* accesses this variable?"

For a properly documented standard utility, the answer to this can be found in the ENVIRONMENT section of the utility's man page. So, for example, the `git` manual

page mentions that `GIT_PAGER` controls which utility is used to browse multi-page output from `git`. Hence,

```
# XXX FIXME: buggy
branch="main"
GIT_PAGER="less"
git log -n 25 --oneline "$branch"
git log "$branch"
```

will not work correctly, because you did not `export GIT_PAGER`. (Of course, if your system already declared the variable as exported somewhere else, the bug is not reproducible.)

We are *explicitly* referring to the variable `$branch`, and the `git` program code doesn't refer to a system variable `branch` anywhere (as also suggested by the fact that its name is written in lower case; but many beginners erroneously use upper case for their private variables, too! See Correct Bash and shell script variable capitalization for a discussion) so there is no reason to `export branch`.

The correct code would look like

```
branch="main"
export GIT_PAGER="less"
git log -n 25 --oneline "$branch"
git log -p "$branch"
```

(or equivalently, you can explicitly prefix each invocation of `git` with the temporary assignment

```
branch="main"
GIT_PAGER="less" git log -n 25 --oneline "$branch"
GIT_PAGER="less" git log -p "$branch"
```

In case it's not obvious, the shell script syntax

```
var=value command arguments
```

temporarily sets `var` to `value` for the duration of the execution of

```
command arguments
```

and exports it to the `command` subprocess, and then afterwards, reverts it back to the previous value, which could be undefined, or defined with a different - possibly empty - value, and unexported if that's what it was before.)

For internal, ad-hoc or otherwise poorly documented tools, you simply have to know whether they silently inspect their environment. This is rarely important in practice, outside of a few specific use cases, such as passing a password or authentication token or other secret information to a process running in some sort of container or isolated environment.

If you really need to know, and have access to the source code, look for code which uses the `getenv` system call (or on Windows, with my condolences, variations like `getenv_s`, `w_getenv`, etc). For some scripting languages (such as Perl or Ruby), look for `ENV`. For Python, look for `os.environ` (but notice also that e.g. `from os import environ as foo` means that `foo` is now an alias for `os.environ`). In Node, look for `process.env`. For C and related languages, look for `envp` (but this is just a convention for what to call the optional third argument to `main`, after `argc` and `argv`; the language lets you call them anything you like). For shell scripts (as briefly mentioned above), perhaps look for variables with uppercase or occasionally mixed-case names, or usage of the utility `env`. Many informal scripts have undocumented but discoverable assignments usually near the beginning of the script; in particular, look for the `?=` default assignment parameter expansion.

For a brief demo, here is a shell script which invokes a Python script which looks for `$NICKNAME`, and falls back to a default value if it's unset.

```
#!/bin/sh
NICKNAME="Handsome Guy"
demo () {
    python3 <<\____
from os import environ as env
print("Hello, %s" % env.get("NICKNAME", "Anonymous Coward"))
____
}
demo
# prints "Hello, Anonymous Coward"
# Fix: forgot export
export NICKNAME
demo
# prints "Hello, Handsome Guy"
```

As another tangential remark, let me reiterate that you only ever need to `export` a variable once. Many beginners cargo-cult code like

```
# XXX FIXME: redundant exports
export PATH="$HOME/bin:$PATH"
export PATH="/opt/acme/bin:$PATH"
```

but typically, your operating system has already declared `PATH` as exported, so this is better written

```
PATH="$HOME/bin:$PATH"
PATH="/opt/acme/bin:$PATH"
```

or perhaps refactored to something like

```
for p in "$HOME/bin" "/opt/acme/bin"
do
    case :$PATH: in
     *:"$p":*) ;;
     *) PATH="$p:$PATH";;
    esac
done
# Avoid polluting the variable namespace of your interactive shell
unset p
```

which avoids adding duplicate entries to your `PATH`.

Share
Improve this answer
Follow

edited Jun 5, 2022 at 12:39

answered Jun 5, 2022 at 12:21

tripleee
189k • 36 • 311 • 359

"typically, your operating system has already declared PATH as exported" -- I believe whatever environment variables that are set in the environment of `bash` when `bash` started will automatically be exported? – user202729 Mar 17, 2023 at 4:44

Yeah, but there are many other shells out there too. – tripleee Mar 17, 2023 at 6:00

0

Although not explicitly mentioned in the discussion, it is NOT necessary to use export when spawning a subshell from inside bash since all the variables are copied into the child process.

Share Improve this answer Follow

answered Sep 4, 2013 at 16:55

Scott
9 • 1

2 Please explain as what you are saying seems to directly contradict the answers w/ examples above. – Mike Lippert Nov 23, 2013 at 16:19

This is the right way if you don't want the variables to be exported globally but only available to the subprocess! Thank you. – WispyCloud Jul 6, 2014 at 2:09

@MikeLippert What Scott means by subshell is the ones created by process substitution $() or ``, subshells created by commands in parentheses (command1; command2) etc automatically inherit all parent shell's variables even if they're not exported. But child processes or scripts invoked won't see all shell variables unless they're exported. This is one of the major differences and is often misunderstood – Pavan May 19, 2021 at 18:24

@Pavan Ah, now that is very helpful. Because a subshell created by invoking a new bash process is NOT what he meant and would only receive exported variables. And that is what I was thinking of when I asked that question so many years ago. – Mike Lippert May 20, 2021 at 21:49

0

Git 2.47 (Q4 2024), batch 1 illustrates another difference between `export VAR` and `VAR=VAL`: `"VAR=VAL shell_func"` should be avoided.

See commit 728a196 (22 Jul 2024) by Junio C Hamano (`gitster`).
(Merged by Junio C Hamano -- `gitster` -- in commit ca9221c, 31 Jul 2024)

> ## `CodingGuidelines`: document a shell that "fails" "VAR=VAL `shell_func"`
>
> Helped-by: Kyle Lippincott
>
> Over the years, we accumulated the community wisdom to avoid the common "one-short export" construct for shell functions, but seem to have lost on which exact platform it is known to fail.
> Now during an investigation on a breakage for a recent topic, we found one example of failing shell.
> Let's document that.
>
> This does *not* mean that we can freely start using the construct once Ubuntu 20.04 is retired.
> But it does mean that we cannot use the construct until Ubuntu 20.04 is fully retired from the machines that matter.
> Moreover, posix explicitly says that the behaviour for the construct is unspecified.

`CodingGuidelines` now includes in its man page:

> The common construct
>
> ```
> VAR=VAL command args
> ```
>
> to temporarily set and export environment variable VAR only while "command args" is running is handy, but this triggers an unspecified behaviour according to POSIX when used for a command that is not an external command (like **shell functions**).

> Indeed, dash 0.5.10.2-6 on Ubuntu 20.04, `/bin/sh` on FreeBSD 13, and AT&T ksh all make a temporary assignment without exporting the variable, in such a case.
>
> As it does not work portably across shells, do not use this syntax for shell functions.
>
> A common workaround is to do an explicit export in a subshell, like so:

```
# (incorrect)
VAR=VAL func args

(correct)
(
  VAR=VAL &&
  export VAR &&
  func args
)
```

> but be careful that the effect "`func`" makes to the variables in the current shell will be lost across the subshell boundary.

Share Improve this answer Follow

answered Aug 17 at 23:43

VonC
1.3m ● 558 ● 4.7k ● 5.6k

See examples in commit c2058b2. – VonC Aug 17 at 23:51

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.