Table of contents

What are environment variables?

The values of environment variables can control the behavior of an operating system, individual utilities such as Git, shell scripts, user applications such as the Google Chrome browser, or deployed applications such as a Python web app.

Let's look at the basics for setting and accessing environment variables using the following commands:

```
# Setting an environment variable

# The `export` keywords essentially means "make
this globally accessible"
# No spaces on either side of the equals sign.
# Quotes recommended
export FULL_NAME="Darth Vader"

# Getting an environment variable's value

# Note the $ prefix used for referencing
echo "Anakin Skywalker is now $FULL_NAME"
# >> Anakin Skywalker is now Darth Vader
```

That's only the tip of the iceberg, so let's dive deeper.

Difference between shell and environment variables

Simplistically, shell variables are local in scope whereas environment variables are global, but let's explore this further with examples.

Shell variables should be used when they are only needed in the current shell or script in which they were defined.

```
# Shell variable as it does not use the `export`
command
WOOKIE="Chewbacca"
echo "The Wookie's name is $WOOKIE"
# >> The Wookie's name is Chewbacca
```

If you opened a new shell and ran the **echo** command, the **NAME** variable does not exist as it was scoped to the previous shell only.

```
# In a new shell
echo "The Wookie's name is $WOOKIE"
# >> The Wookie's name is
```

The same rules apply to scripts, for example, if a file **shell-var-test.sh** contained the following:

```
# shell-var-test.sh
echo "The Wookie's name is $WOOKIE"
```

And **shell-var-test.sh** was run even after **NAME** was defined, it's not accessible to the script.

```
WOOKIE="Chewbacca"
bash shell-var-test.sh
# >> The Wookie's name is
```

We can also verify this behavior this using a non-shell language such as Python:

```
WOOKIE="Chewbacca"
python -c "import os;
print(os.environ.get('WOOKIE'))"
# >> None
```

Environment variables, on the other hand, are designed to be accessible to scripts or child processes and differ from shell variables by use of the **export** command.

```
export WOOKIE="Chewbacca"
bash shell-var-test.sh
# >> The Wookie's name is Chewbacca
```

It's also true that any variable changes inside a script or new process do not affect the shell where they were executed from, even for environment variables.

We delve deeper into the scope of environment variables in shells and shell scripts later in this article, as well as how to change the default scoping behavior by learning how to execute a script in the context of the current shell.

How to change and set environment variables

Changing an environment variable is no different from changing a shell variable:

```
export FAV_JEDI="Obi-Wan Kenobi"
echo "My favorite Jedi is $FAV_JEDI"
# >> My fav jedi is Obi-Wan Kenobi

FAV_JEDI="Rey Skywalker"
echo "My favorite Jedi is now $FAV_JEDI"
# >> My favorite Jedi is now Rey Skywalker
```

You can also modify a variable using its original value to create a new value. You've most likely seen this used before in a **~/.bash_profile** or **~/.bashrc** file when appending a directory to the **$PATH** variable:

```
# Add the `~/bin` directory to $PATH for user
scripts and self-compiled binaries
PATH=$PATH:~/bin
```

Note that we **did not** put **export** before **PATH** in this example, and that's because **PATH** was already exported, so putting **export** again does nothing.

How to delete an environment variable

The **unset** command is used to remove a variable:

```
export FAV_DROID="R2-D2"
echo "My favorite droid is $FAV_DROID"
# >> My favorite droid is R2-D2

unset FAV_DROID
echo "My favorite droid is $FAV_DROID"
# >> My favorite droid is
```

The deletion of an environment variable only affects the current shell.

Environment variable syntax and best practices

Here are some rules and best practices for assigning and using environment variables:

- Variable names may only contain characters (a-z,A-Z), numbers, and underscores
- No spaces on either side of the equals sign
- The naming convention is **CAPITALIZED_SNAKE_CASE**
- Always surround values with quotes, e.g. **export KEY="value"**
- Always surround variables usage with quotes, e.g. **echo "$KEY"**
- Only export a variable if it will be used by script called from that shell or child process
- Avoid exporting variables that contain sensitive information such as API tokens by setting them for a single command only.

How to capture the output of a command and assign it to an environment variable

It's common to capture the output of a command using command substitution **$(...)** and assign it to a variable. For example, assign the value of a date formatted string:

```
TODAYS_DATE=$(date  +"%B %-d, %Y")
echo "Today's date is $TODAYS_DATE"
```

Using double or single quotes for environment variables

Using double or single quotes depends on your requirements and as a rule, always quote your values to avoid unintended word splitting issues:

```
export FULL_NAME=Obi-Wan Kenobi
# >> bash: Kenobi: command not found

FULL_NAME="Obi-Wan Kenobi"
echo "Help me $FULL_NAME. You're my only hope."
# >> Help me Obi-Wan Kenobi... You're my only hope
```

Double quotes allow the use of variable referencing and command substitution **$(...)**, for example, getting the current username:

```
echo "My username is $(whoami) and I am currently
in the $PWD directory"
# >> My username is ryan and I am currently in the
/home/ryan/dev directory
```

If using double quotes for variable referencing but also want to output the $ sign, then you need to escape it using the backslash \ character:

```
export TRAVELLERS="Luke and Obi-Wan"
echo "Congratulations $TRAVELLERS, you've just won
passage on the Millenium Falcon worth \$17,000"
# >> Congratulations Luke and Obi-Wan, you've just
won passage on the Millenium Falcon worth $17,000
```

Another option instead of escaping a reserved symbol such as $ is using single quotes, as it prevents variable referencing and command substitution because the string is not interpolated:

```
echo 'The $PWD env var is the path to the current
working directory'
# >> The $PWD env var is the path to the current
working directory
```

Finally, if you're in a situation where the **$VAR_NAME** form doesn't work because of ambiguity, then you need the **${VAR_NAME}** syntax instead:

```
export FOO="foo"
echo "$FOObar" # Output will be empty as variable
`$FOObar` does not exist
echo "${FOO}bar"
# >> foobar
```

Scope of environment variables in shells and shell scripts

Every command, script, and application runs in its own process, each having a unique identifier (typically referred to as **PID**). It may run for a few milliseconds (e.g. **ls -la**), or many hours, e.g. Visual Studio Code or a Python application server.

To see this in action, let's run a **sleep** command as a background process by appending **&** after the command so we see its PID in the shell:

```
# Run a command as a background process using `&`
to see the PID of that command
sleep 5 &
```

When running a command or script from the shell, the current shell is the **parent process** and the command or script is a **child process**, which only has access to the environment variables from the current shell or parent process.

For security and isolation, any modifications to environment variables in a child process do not affect the parent process or any other shell sessions.

We can demonstrate this using the subshell **(...)** syntax or executing a string as a bash command:

```
# Subshell example
(export SUBSHELL_VAR='Test') && echo $SUBSHELL_VAR
# Empty output

# Child shell example by executing string with
bash
bash -c "export SUBSHELL_VAR='Test'" && echo
$SUBSHELL_VAR
# Empty output
```

In both instances, the **SUBSHELL_VAR** environment variable lived and died within the child process in which it was created and was not visible to the parent process.

It can be mind-blowing when you realize every single command, script, or application is a process, all the way up to PID 1 which is the process from which every other child process inherits from, even if not directly. You can use the **pstree** command (For Mac users: **brew install pstree**) to visually see this hierarchy in action.

Setting environment variables for a single command

Sometimes, you'll want to expose or change environment variables only for the life of a single command, script, or child process.

To demonstrate, we'll use the following script **baby-yoda.sh**:

```
# baby-yoda.sh
echo "Baby Yoda is named $BABY_YODA"
```

First, let's look at how to do this the long and inefficient way:

```
export BABY_YODA="Grogu"
bash baby-yoda.sh
# >> Baby Yoda is named Grogu
unset BABY_YODA
```

We can instead turn this into a single command by defining the variable before command:

```
BABY_YODA="Grogu" bash baby-yoda.sh
# >> Baby Yoda is named Grogu
```

Now you might be wondering what if you want to temporarily expose an environment variable to multiple commands? It may not work the way you would expect. Let's try running our script twice using **&&** syntax:

```
BABY_YODA="Grogu" bash baby-yoda.sh && bash baby-
yoda.sh
# >> Baby Yoda is named Grogu
# >> Baby Yoda is named
```

The second script invocation did not have the **BABY_YODA** environment variable set because the shell interpreted the above command as:

```
BABY_YODA="Grogu" bash baby-yoda.sh
bash baby-yoda.sh
```

To temporarily expose an environment variable to multiple commands, we can pass a string to bash to execute:

```
BABY_YODA="Grogu" bash -c 'bash baby-yoda.sh &&
bash baby-yoda.sh'
# >> Baby Yoda is named Grogu
# >> Baby Yoda is named Grogu
```

How to check if an environment variable exists

At some point, you'll need to conditionally check for the existence of an environment variable.

In most situations, you'll want to check a variable is set with a non-empty value so that's what we'll start off with. To demonstrate, save the following code to **deathstar-attack.sh**:

```
# deathstar-attack.sh
if [ -z "$DEATHSTAR_SHIELD_DOWN" ]; then
    echo "Break off the attack! The shield is
still up!"
else
    echo "The shield is down! Commence attack on
the Death Star's main reactor!"
fi
```

Then let's execute the script without defining the **DEATHSTAR_SHIELD_DOWN** variable:

```
bash deathstar-attack.sh
# >> Break off the attack! The shield is still up!
```

Now with defining **DEATHSTAR_SHIELD_DOWN**:

```
export DEATHSTAR_SHIELD_DOWN="Yes"
bash deathstar-attack.sh
# >> The shield is down! Commence attack on the
Death Star's main reactor!
```

If on the other hand, you only want to check if a variable is set and don't care if it's empty or not, then use the following, saving it to **death-star-attack-2.sh**:

```
# deathstar-attack-2.sh
if [ -z "${DEATHSTAR_SHIELD_DOWN+x}" ]; then
    echo "Break off the attack! The shield is
still up!"
else
    echo "The shield is down! Commence attack on
the Death Star's main reactor!"
```

```
fi
```

This works because the **${DEATHSTAR_SHIELD_DOWN+x}** uses parameter expansion which for this purpose, evaluates to nothing if **DEATHSTAR_SHIELD_DOWN** is unset.

```
unset DEATHSTAR_SHIELD_DOWN
bash deathstar-attack-2.sh
# >> Break off the attack! The shield is still up!

export DEATHSTAR_SHIELD_DOWN=""
bash deathstar-attack-2.sh
# >> The shield is down! Commence attack on the
Death Star's main reactor!
```

Where to set environment variables

Where and how you set environment variables depends on the operating system and whether you want to set them at a system or individual user level, as well as what shell you're using, e.g. **bash** or **zsh** (which is now the default in Mac).

To avoid simply repeating content for the sake of it, check out this excellent comprehensive article from Unix/Linux Stack Exchange question: How to permanently set environmental variables.

How to list environment variables

The **printenv** command does exactly as it describes, printing out all environment variables available to the current shell.

```
printenv
```

You can combine **printenv** with **grep** to filter the list environment variables:

```
printenv | grep GIT
```

To output the value of a specific environment variable, you should use also **printenv**:

```
printenv HOME
```

You might be thinking "can't you use **echo** for that"? And yes, you can, but **echo** works for shell variables too which makes **printenv** superior, because it only works for environment variables.

If you're wondering how to tell if **printenv SOME_VAR** actually worked or whether **SOME_VAR** was just empty, you can inspect the **$?** variable after the **printenv command** and if the exit code was 0, the environment variable exists.

How to list all shell and environment variables

It's possible to list both shell and environment variables using the set command:

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

You'll notice that you see not just variables, but functions too.

If using bash, you can use **compgen** which is less noisy**:**

```
compgen -A variable
```

Common environment variables in Linux and Mac

The following environment variables are not an exhaustive list, but are the most common between Linux and Mac:

- **$PATH:** A colon-separated list of directories for looking up binaries and scripts that don't need the full path in order to execute them, e.g. **echo** instead of **/bin/echo** as **/bin** is in $**PATH**
- **$HOME:** The user's home directory for the current shell
- **$PWD:** The directory location in the current shell
- **$SHELL:** The executable for the current shell, e.g. **/bin/bash**
- **$HOSTNAME:** The hostname of the current machine as it may appear in your local network

Using environment variables for application secrets and credentials

Environment variables are arguably the best way to supply config and secrets to deployed applications such as a Python or Node.js app.

This is because Virtual Machines (VM) and Docker containers provide a sandboxed environment for applications to execute in, therefore any environment variable only affects that specific application in that VM or container.

Because applications are configured differently from development to production, configuring an app using environment variables means only the values need to change when deploying to different environments, not the source code.

Environment variables should also be used to configure CI/CD jobs and environments, e.g, GitHub Actions.

Supplying application config and secrets using environment variables is precisely what the Doppler CLI does, and our mission is to make this as fast, easy, and secure as possible with integrations for every major every cloud provider and platform.

How to execute a script in the context of the current shell

Most times, you'll want a script executed in a child process that can access, but can't modify the current shell. But there are scenarios where you'll want to load functions or variables from a script into the current shell.

An example is the bash autocompletion code from homebrew on macOS using the leading period syntax:

```
# Load bash autocompletion code into the current shell
. /usr/local/Cellar/bash-completion/1.3_3/etc/bash_completion
```

Let's demonstrate this functionality by loading variables into the current shell using the below script, saving it as **current-shell-vars.sh:**

```
#!/usr/bin/env bash
```

```
COOL_DROID="R2-D2"
ANNOYING_DROID="C-3PO"
```

Then make the script executable:

```
chmod +x current-shell-vars.sh
```

Simply executing the script will only make the variables available to the code inside the script:

```
./current-shell-vars.sh
echo $COOL_DROID
# empty output
```

To execute it in the current shell, we prefix the command with a leading period:

```
. ./current-shell-vars.sh
echo "$COOL_DROID is cool but $ANNOYING_DROID is
pretty annoying"
# >> R2-D2 is cool but C-3PO is pretty annoying
```

Be very careful with this functionality and only execute scripts you absolutely trust!

If you're wondering, "doesn't **source** do the same thing?", yes it does, but it's only available in bash, so the above form is simply more portable

How to pass through environment variables when using sudo

By default for security, commands run as root using **sudo** do not pass through environment variables from the current shell, but you can override this using the **--preserve-env** flag:

```
# Print the environment variables for the root
user
sudo printenv

# Pass through the environment variables from the
current shell
sudo --preserve-env printenv
```

Use this caution!

How to execute a command or script in a "clean" environment

Sometimes, you'll want to execute a command or script in a "clean" environment, devoid of all environment variables using **env -i**:

```
env -i printenv
# Empty output as no env vars exist
```

How to set an environment variable to be read-only

To make an environment variable (or function) read-only, just use the **readonly** command:

```
readonly export READONLY_VAR=1
READONLY_VAR=2
# >> READONLY_VAR: readonly variable
```

Additional resources

There's obviously a lot more to shell programming than just environment variables and here are three essential resources to take your shell usage and scripts to the next level: