

Forum:[Hack Forums](#)

Thread: [Running your scripts anywhere](#)

SIM-3429335 - December 27, 2018, 3:05 am

So I just came across this really cool thing so I learned you can run a program you're working on in Linux from the terminal with out being in the directory the program is in so if I type

```
root@local/: helloworldscript
```

The script from which dir it's in will run and the return will be

```
root@local/: helloworldscript
Hello world
root@local/:
```

You can do this by using links here's how you do it (I usually put my custom scripts in the /opt dir

```
In -s /opt/scripts/helloworldscript.py /usr/bin/helloworldscript
```

So what's going on is that you're basically creating a symbolic link that the system can use as a command

In - link command

-s - creates a symbolic link

/opt/scripts/helloworldscript.py - Specify the directory where the program is located(use the entire name of the program including extension)

/usr/bin/helloworldscript - the directory where it will be run (usr/bin is kinda like sys32 that's where commands are recognized from) and the name "helloworldscript" you can call the command whatever you like just as long as the directory and your /usr dir is correct.

Happyhunting ?

Asdethprime - December 27, 2018, 6:10 am

That's one way of tackling this problem; not the only way.

Let's take a step back. Why is it that when you go on a bash terminal and you type
\$ clear
and it runs from anywhere but you do the same thing for your program it doesn't?

The answer to that lies in the environment variable PATH. As an exercise, go to your terminal and try this:

```
$ echo $PATH
```

NOTE: individual path are separated with ":",

You are going to get a list of folders that the system will check in order to search for this.

You can picture what is happening in bash as follows:

1) bash asks if the command typed is something native - a command defined within bash binary (/bin/bash). If so then it knows how to execute it and hence goes ahead and executes it.
2) if the command is NOT native to bash, it will look for files in the PATH variable, in the order that is listed.

You make sure you can run your code from anywhere in the command line, putting a symbolic link to it on opt folder is 1 idea. This works because opt folder happens to be in the path.

This can be tackled in different ways, for example, you could simply add your program's full path to the \$PATH variable itself. This way you do not have to create a symbolic link [though symbolic links don't consume a lot of memory or come with overhead].

```
$ export PATH="$PATH:/usr/bin/helloworldscript"
```

This should achieve the same result.

Don't just listen to me go try it =)
WARNING

export function is NOT permanent.

Once you exit bash, the exported part no longer has effect.

You can automatically get bash to export whatever you want to export by editing its configuration variable.

SIM-3429335 - December 27, 2018, 12:17 pm

(Dec 27, 2018 07:10 AM)Asdethprime Wrote: [\[CITE\]](#) That's one way of tackling this problem; not the only way.

Let's take a step back. Why is it that when you go on a bash terminal and you type
\$ clear
and it runs from anywhere but you do the same thing for your program it doesn't?

The answer to that lies in the environment variable PATH. As an exercise, go to your terminal and try this:
\$ echo \$PATH

NOTE: individual path are separated with ":",

You are going to get a list of folders that the system will check in order to search for this.

You can picture what is happening in bash as follows:

1) bash asks if the command typed is something native - a command defined within bash binary (/bin/bash). If so then it knows how to execute it and hence goes ahead and executes it.
2) if the command is NOT native to bash, it will look for files in the PATH variable, in the order that is listed.

You make sure you can run your code from anywhere in the command line, putting a symbolic link to it on opt folder is 1 idea. This works because opt folder happens to be in the path.

This can be tackled in different ways, for example, you could simply add your program's full path to the \$PATH variable itself. This way you do not have to create a symbolic link [though symbolic links

don't consume a lot of memory or come with overhead].

```
$ export PATH="$PATH:/usr/bin/helloworldscript"
```

This should achieve the same result.

Don't just listen to me go try it =)

WARNING

export function is NOT permanent.

Once you exit bash, the exported part no longer has effect.

You can automatically get bash to export whatever you want to export by editing its configuration variable.

Haven't tried it that way lol I just found out about this when I was messing around in Debian I'll try it and see how it goes

sh4moon - December 27, 2018, 1:25 pm

I just happened to see one time that a shell script in /usr/bin just switched to the directory the program was in and ran it. it might have been thefatrat. it looked like this.

```
Code #!/bin/bash
cd /path/to/program
./program
```

Asdethprime - December 28, 2018, 2:30 am

(Dec 27, 2018 01:17 PM)fraggmaster10 Wrote: [\[CITE\]](#) Haven't tried it that way lol I just found out about this when I was messing around in Debian I'll try it and see how it goes
Definitely worth giving this a try.

Be warn though, I was not trying to give you a better method.

Flat out, creating a symbolic link (your method) is permanent and will work till the said link is removed.

The export PATH method is not permanent, but that was just to give you an idea of how things work under the hood

(Dec 27, 2018 02:25 PM)sh4moon Wrote: [\[CITE\]](#) I just happened to see one time that a shell script in /usr/bin just switched to the directory the program was in and ran it. it might have been thefatrat. it looked like this.

```
Code #!/bin/bash
cd /path/to/program
./program
```

That's actually slightly different:

```
Code #!/bin/bash
cd /path/to/program
./program
```

Code like this is normally added so the program itself finds where its sub-component is. Once the program runs, no matter where you are, the program will now be able to find where your remaining code is.

I like long complicated answer hehe so brace yourself!!!! :p

But if this bash script is located in say ~/my_sexy_folder but you try and run this code from just ~ ; bash won't be able to find the script. This is because you have not told bash where it is. For Bash to move into your program directory, it needs to find this script and need to reach in your code the second line of execution (that's counting kernel picking the interpreter as 1 line of code - the #! statement).

To reach there you need to be on the system path.

But this works wonderfully well if you have a framework that already has a global presence [i.e. has at least 1 command that you can execute from anywhere] ; but the program you create is local and sits in the particular folder. For example: metasploit and thefatrat ; or other places you see this will be emberjs framework (for javascript developers) or django framework (if you are like me, enjoy python and find Flask a bit hard to get started with hehe).

This works because here bash does not have to find the relevant command line binary. So for TheFatRat framework it would be the thefatrat, for emberjs it would be ember, for django it would be django-admin. Once command line find these than you can have this code on configuration for the respective program.

Now what happens:

- 1) Bash finds the command because its already on the path
- 2) This starts the framework but how is the framework going to find where your program is; well it normally stores codes like these on its configuration settings.
- 3) the framework will look into the settings - move into your program - then it no longer has to worry about finding it