a loop in Bash

a set of actions on multiple files with for loops and find commands.

12 Jun 2019 | Seth Kenlon (Red Hat) (/users/seth) | 143 | 10 comments

A common reason people want to learn the Unix shell is to unlock the power of batch processing. If you want to perform some set of actions on many files, one of the ways to do that is by constructing a command that iterates

g terminology, this is called *execution control,* xamples of it is the **for** loop.

*w*hat actions you want your computer to take file) you specify.

Linux (https://opensource.com/life/17/10/top-7016000000127cYAAQ)

- 10 command-line tools for data analysis in Linux (https://opensource.com/article/17/2/command-line-tools-data-analysis-linux?intcmp=7016000000127cYAAQ)
- Download Now: SSH cheat sheet (https://opensource.com/downloads/advanced-ssh-cheat-sheet?intcmp=7016000000127cYAAQ)
- Advanced Linux commands cheat sheet (https://developers.redhat.com/cheat-sheets/advanced-linux-commands/?intcmp=7016000000127cYAAQ)
- Linux command line tutorials (https://opensource.com/tags/command-line?intcmp=7016000000127cYAAQ)

An easy loop to try is one that analyzes a collection of files. This probably isn't a useful loop on its own, but it's a safe way to prove to yourself that you

files (such as JPEG, PNG, or similar). You can create the folder and copy files into it using a file manager or in the terminal:


```
tion/*.{png,jpg} example
```

older, then list the files in it to confirm that ou expect:

The syntax to loop through each file individually in a loop is: create a variable (**f** for file, for example). Then define the data set you want the variable to cycle through. In this case, cycle through all files in the current directory using the * wildcard character (the * wildcard matches *everything*). Then terminate this introductory clause with a semicolon (**;**).

```
$ for f in * ;
```

Depending on your preference, you can choose to press **Return** here. The shell won't try to execute the loop until it is syntactically complete.

Next, define what you want to happen with each iteration of the loop. For simplicity, use the **file** command to get a little bit of data about each file, represented by the **f** variable (but prepended with a **$** to tell the shell to

```
do file $f ;
```

Terminate the clause with another semi-colon and close the loop:

cycling through *everything* in the current
each file, one by one, to the variable **f** and

```
                    data, EXIF standard 2.2
                    G image data, 4608 x 2592, 8-bit/color RGB
otago.jpg: JPEG image data, EXIF standard 2.2
waterfall.png: PNG image data, 4608 x 2592, 8-bit/color RGB,
```

You can also write it this way:

```
$ for f in *; do file $f; done
cat.jpg: JPEG image data, EXIF standard 2.2
design_maori.png: PNG image data, 4608 x 2592, 8-bit/color RGB
otago.jpg: JPEG image data, EXIF standard 2.2
waterfall.png: PNG image data, 4608 x 2592, 8-bit/color RGB,
```

Both the multi-line and single-line formats are the same to your shell and produce the exact same results.

A practical example

send to friends. Your photo files are huge, making them too large to email and inconvenient to upload to your photo-sharing service (http://nextcloud.com). You want to create smaller web-versions of your

os and don't want to spend the time reducing

command using your package manager on ce, on Fedora and RHEL:

```
ck
```

```
ck
```

On BSD, use **ports** or pkgsrc (http://pkgsrc.org). On Mac, use Homebrew (http://brew.sh) or MacPorts (https://www.macports.org).

Once you install ImageMagick, you have a set of new commands to operate on photos.

Create a destination directory for the files you're about to create:

```
$ mkdir tmp
```

To reduce each photo to 33% of its original size, try this loop:

```
$ for f in * ; do convert $f -scale 33% tmp/$f ; done
```

You can use any number of commands within a loop, so if you need to perform complex actions on a batch of files, you can place your whole workflow between the **do** and **done** statements of a **for** loop. For example,

X ı processed photo straight to a shared photo
remove the photo file from your local

```
p/$f
eth@example.com:~/public_html
```

for loop, your computer automatically runs
f you process just 10 photos this way, you
d probably at least as many minutes.

Limiting your loop

A loop doesn't always have to look at every file. You might want to process only the JPEG files in your example directory:

```
$ for f in *.jpg ; do convert $f -scale 33% tmp/$f ; done
$ ls -m tmp
cat.jpg, otago.jpg
```

Or, instead of processing files, you may need to repeat an action a specific number of times. A **for** loop's variable is defined by whatever data you provide it, so you can create a loop that iterates over numbers instead of files:

1
2
3
4

e your own loops. Until you're comfortable
es of the files you want to process and, as
ds with built-in safeguards to prevent you
making irreparable mistakes, like accidentally
files to the same name, each overwriting the

ead on.

Not all shells are Bash

The **for** keyword is built into the Bash shell. Many similar shells use the same keyword and syntax, but some shells, like tcsh (https://en.wikipedia.org/wiki/Tcsh), use a different keyword, like **foreach**, instead.

In tcsh, the syntax is similar in spirit but more strict than Bash. In the following code sample, do not type the string **foreach?** in lines 2 and 3. It is a secondary prompt alerting you that you are still in the process of building your loop.

```
$ foreach f (*)
```

```
otago.jpg: JPEG image data, EXIF standard 2.2
waterfall.png: PNG image data, 4608 x 2592, 8-bit/color RGB, non-inte
```


x must appear alone on separate lines, so you
e line as you can with Bash and similar

nd command

that doesn't provide a **for** loop function, or
ferent command with added features.

way to implement the functionality of a **for**
o define the scope of which files to include in
Parallel (https://opensource.com/article/18/5/gnu-parallel) processing.

The **find** command is meant to help you find files on your hard drives. Its syntax is simple: you provide the path of the location you want to search, and **find** finds all files and directories:

```
$ find .
.
./cat.jpg
./design_maori.png
./otago.jpg
./waterfall.png
```

You can filter the search results by adding some portion of the name:

The great thing about **find** is that each file it finds can be fed into a loop using the **-exec** flag. For instance, to scale down only the PNG photos in your example directory:


```
convert {} -scale 33% tmp/{} \;

png
```

t characters **{}** stand in for whatever item
ds, any file ending in PNG that has been
ec clause must be terminated with a
es to use the semicolon for itself. You
backslash (**\;**) so that **find** knows to treat that
aracter.

The **find** command is very good at what it does, and it can be too good sometimes. For instance, if you reuse it to find PNG files for another photo process, you will get a few errors:

```
$ find . -name "*png" -exec convert {} -flip -flop tmp/{} \;
convert: unable to open image `tmp/./tmp/design_maori.png':
No such file or directory @ error/blob.c/OpenBlob/2643.
...
```

It seems that **find** has located all the PNG files—not only the ones in your current directory (**.**) but also those that you processed before and placed in your **tmp** subdirectory. In some cases, you may want **find** to search the current directory plus all other directories within it (and all directories in *those*). It can be a powerful recursive processing tool, especially in complex file structures (like directories of music artists containing directories of

To find only PNG files in the current directory (excluding subdirectories):

```
$ find . -maxdepth 1 -name "*png"
```


current directory plus an additional level of
aximum depth by 1:

```
'*png"
```

subdirectories.

profit

ore time and effort you save, and the bigger the tasks you can tackle. You're just one user, but with a well-thought-out loop, you can make your computer do the hard work.

You can and should treat looping like any other command, keeping it close at hand for when you need to repeat a single action or two on several files. However, it's also a legitimate gateway to serious programming, so if you have to accomplish a complex task on any number of files, take a moment out of your day to plan out your workflow. If you can achieve your goal on one file, then wrapping that repeatable process in a **for** loop is relatively simple, and the only "programming" required is an understanding of how variables work and enough organization to separate unprocessed from processed files. With a little practice, you can move from a Linux user to a Linux user who knows how to write a loop, so get out there and make your computer work for you!

(/article/19/10/programming-bash-loops)

ash: Loops (/article/19/10/programming-

ing iterative operations, in the final article in this with Bash.

both)

e (/tags/command-line) **Linux** (/tags/linux)

(/users/seth)

About the author

Seth Kenlon - Seth Kenlon is an independent multimedia artist, free culture advocate, and UNIX geek. He has worked in the film (http://www.imdb.com/name/nm1244992) and computing (http://people.redhat.com/skenlon) industry, often at the same time. He is one of the maintainers of the Slackware-based multimedia production project, http://slackermedia.info (http://slackermedia.info)

• More about me (/users/seth)

asousa) on 12 Jun 2019

0

eginner's guide to SH for remote nection on Linux (/article amm) on 12 Jun 2019

Program hardware from the Linux command line (/article /20/9/hardware- 2

y small bits of valuable information. This "for loop" to increase productivity when used in correct coding on Gonca Sousa

son?utm_campaign=intr) command- line?utm_campaign=intr

owtopamm) on 12 Jun 2019

0

y @Seth Kenlon. You are the author and I quoted Gonca Sousa in my thanks for the lesson statement in the above comment. Sorry Seth Kenlon and to Gonca Sousa as well for the incorrect credit. howtopamm

A practical guide to learning awk (/article /20/9/awk-ebook?utm_campaign=intr)

Open ports and route traffic through your firewall (/article /20/9/firewall?utm_campaign=int)

An introduction to using tcpdump at the Linux command line (/article/18/10 /introduction-tcpdump?utm_campaig

Seth Kenlon (/users/seth) on 12 Jun 2019 0

Thanks for reading my article and for the comment!

JJ (/users/wavesailor) on 12 Jun 2019 0

Great article - Nice to learn about the `file` command and the `covert` in addition to learning about looping. Thx

0

open every image I resize or convert from one format to another in a GUI, one at a time.

Thanks for reading, JJ, and thanks for the comment!

kritiko) on 19 Jun 2019 0

++ To the author for the idea and execution.

seth) on 19 Jun 2019 1

ve to be ((++))

Carl T. Miller (/users/carltm) on 25 Jun 2019 1

Great article! I use the type of looping you describe.

I notice that you didn't cover the case where you have hundreds or even thousands of files to process, and you want them run sequentially instead of in parallel. The following command can be rewritten easily.

```
find . -name "*png" -exec convert {} -scale 33% tmp/{} \;
```

This is the new version.

```
find . -name "*png" |\
while IFS= read -r file; do
convert "$file" -scale 33% tmp/
done
```

rush past the intricacies of analyzing find's output and what you need to set IFS to (or what IFS is to begin with), and how read needs to be set, for the input to be sane.

X te article, though. I've taken note of it, but if you'd like to slide this URL over to you...
ı/how-submit-article (https://opensource.com
e)