

For and Read-While Loops in Bash

How to loop, aka designing a program to do repetitive work for you

The loop is one of the most fundamental and powerful constructs in computing, because it allows us to repeat a set of commands, as many times as we want, upon a list of items of our choosing. Much of computational thinking involves taking one task and solving it in a way that can be applied repeatedly to all other similar tasks, and the for loop is how we make the computer do that repetitive work:

```
for item in $items
do
    task $item
done
```

Unlike most of the code we've written so far at the interactive prompt, a for-loop doesn't execute as soon as we hit **Enter**:

```
user@host:~$ for item in $items
```

We can write out as many commands as we want in the **block** between the `do` and `done` keywords:

```
do
    command_1
    command_2
    # another for loop just for fun
    for a in $things; do; command_3 a; done
    command_4
done
```

Only until we reach `done`, and hit **Enter**, does the for-loop do its work.

This is fundamentally different than the line-by-line command-and-response we've experienced so far at the prompt. And it presages how we will be programming further on: less emphasis on executing commands with each line,

and more emphasis on *planning* the functionality of a program, and then *executing* it later.

Basic syntax

The syntax for `for` loops can be confusing, so here are some basic examples to prep/refresh your comprehension of them:

```
for animal in dog cat 'fruit bat' elephant ostrich
do
    echo "I want a $animal for a pet"
done
```

Here's a more elaborate version using variables:

```
for thing in $collection_of_things
do
    some_program $thing
    another_program $thing >> data.txt
    # as many commands as we want
done
```

A command substitution can be used to generate the items that the `for` loop iterates across:

```
for var_name in $(seq 1 100); do
    echo "Counting $var_name..."
done
```

If you need to read a list of lines from a file, and are absolutely sure that none of the lines contain a space within them:

```
for url in $(cat list_of_urls.txt); do
    curl "$url" >> everywebpage_combined.html
done
```

A **read-while** loop is a variation of the above, but is safer for reading lines from a file:

```
while read url
do
    curl "$url" >> everywebpage_combined.html
done < list_of_urls.txt
```

Constructing a basic for loop

Let's start from a beginning, with a very minimal `for` loop, and then built it into something more elaborate, to help us get an understanding of their purpose.

The simplest loop

This is about as simple as you can make a **for** loop:

```
user@host:~$ for x in 1
> do
> echo Hi
> done
Hi
```

Did that seem pretty worthless? Yes it should have. I wrote four lines of code to do what it takes a single line to do, `echo 'Hi'`.

More elements in the collection

It's hard to tell, but a "loop" did execute. It just executed *once*. OK, so how do we make it execute more than one time? Add more (space-separated) elements to the right of the `in` keyword. Let's add four more `1`'s:

```
user@host:~$ for x in 1 1 1 1
> do
> echo Hi
> done
Hi
Hi
Hi
Hi
```

OK, not very exciting, but the program definitely seemed to at least *loop*: four `1` 's resulted in four `echo` commands being executed.

What happens when we replace those four `1` 's with different numbers? And maybe a couple of words?

```
user@host:~$ for x in Q Zebra 999 Smithsonian
> do
> echo Hi
> done
Hi
Hi
Hi
Hi
```

And...*nothing*. So the *loop* doesn't automatically *do* anything *specific* to the collection of values we gave it. Not yet anyway.

Refer to the loop variable

Let's look to the *left* of the `in` keyword, and at that `x`. What's the point of that `x`? A lowercase `x` isn't the name of a keyword or command that we've encountered so far (and executing it alone at the prompt will throw an error). So maybe it's a variable? Let's try referencing it in the `echo` statement:

```
user@host:~$ for x in Q Zebra 999 Smithsonian
> do
> echo Hi $x
> done
Hi Q
Hi Zebra
Hi 999
Hi Smithsonian
```

Bingo. This is pretty much the fundamental workings of a `for` loop: - Get a collection of items/values (`Q Zebra 999 Smithsonian`) - Pass them into a `for` loop construct - Using the loop variable (`x`) as a placeholder, write commands between the `do / done` block. - When the loop executes, the loop variable, `x`, takes the value of each of the items in the list – `Q`, `Zebra`, `999`, `Smithsonian`, – and the block of commands between `do` and `done` is then executed. This sequence repeats once for every item in the list.

The `do / done` block can contain any sequence of commands, even another `for` - loop:

```
user@host:~$ for x in Q Zebra 999 Smithsonian
> do
> echo Hi
> done
Hi Q
Hi Zebra
Hi 999
Hi Smithsonian
```

```
user@host:~$ for x in $(seq 1 3); do
>   for y in A B C; do
>     echo "$x:$y"
>   done
> done
1:A
1:B
1:C
2:A
2:B
2:C
3:A
3:B
3:C
```

Loops-within-loops is a common construct in programming. For the most part, I'm going to try to avoid assigning problems that would involve this kind of logic, as it can be tricky to untwist during debugging.

Read a file, line-by-line, reliably with read-while

Because `cat` prints a file line-by-line, the following for loop seems sensible:

```
user@host:~$ for line in $(cat list-of-dirs.txt)
> do
>   echo "$line"
> done
```

However, the command substitution will cause `cat` to split words by space. If `list-of-dirs.txt` contains the following:

```
Apples
Oranges
Documents and Settings
```

The output of the `for` loop will be this:

```
Apples
Oranges
Documents
and
Settings
```

A **read-while** loop will preserve the words within a line:

```
user@host:~$ while read line
do
    echo "$line"
done < list-of-dirs.txt
Apples
Oranges
Documents and Settings
```

We can also pipe from the result of a command by enclosing it in `<(and)`:

```
user@host:~$ while read line
do
    echo "Word count per line: $line"
done < <(cat list-of-dirs.txt | wc -w)
1
1
3
```

Pipes and loops

If you're coming from other languages, data streams may be unfamiliar to you. At least they are to me, as the syntax for working with them is far more direct and straightforward in Bash than in Ruby or Python.

However, if you're new to programming in any language, what might also be unclear is how working with data streams is different than working with loops.

For example, the following snippet:

```
user@host:~$ echo "hello world i am here" | \
> tr '[:lower:]' '[:upper:]' | tr ' ' '\n'
HELLO
WORLD
I
AM
HERE
```

– produces the same output as this loop:

```
for word in hello world i am here; do
    echo $word | tr '[:lower:]' '[:upper:]'
done
```

And depending on your mental model of things, it does seem that in both examples, each word, e.g. `hello`, `world`, is passed through a process of translation (via `tr`) and then echoed.

Pipes and filters

Without getting into the fundamentals of the Unix system, in which a pipe operates fundamentally different than a loop here, let me suggest a mental workaround:

Programs that pipe from `stdin` and `stdout` can usually be arranged as filters, in which a stream of data goes into a program, and comes out in a different format:


```
# send the stream through a reverse filter
user@host:~$ echo "hello world i am here" | rev
ereh ma i dlrow olleh

# filter out the first 2 characters
user@host:~$ echo "hello world i am here" | cut -c 3-
llo world i am here

# filter out the spaces
user@host:~$ echo "hello world i am here" | tr -d ' '
helloworldiamhere

# filter out words with less than 4 characters
user@host:~$ echo "hello world i am here" | grep -oE '[a-z]{4,}'
hello
world
here
```

For tasks that are more than just transforming data, from filter to filter, think about using a loop. What might such as a task be? Given a list of URLs, download each, and email the downloaded data, with a customized body and subject:

```
user@host:~$ while read url; do
    # download the page
    content=$(curl -Ls $url)
    # count the words
    num_of_words=$(echo $content | wc -w)
    # extract the title
    title=$(echo $content | grep -oP '(?<=<title>)[^<]+')
    # send an email with the page's title and word count
    echo "$content" | mail whoever@stanford.edu -s "$title: $num_of_
words words"
    echo "...Sending: $title: $num_of_words words"
done < urls.txt
```

The data input source, each URL in `urls.txt`, isn't really being *filtered* here. Instead, a multi-step task is being done for each URL.

Piping into read-while

That said, a loop itself can be implemented as just one more filter among filters. Take this variation of the read-while loop, in which the result of `echo | grep` is piped, line by line, into the `while` loop, which prints to stdout using `echo`, which is redirected to the file named `some.txt`:

```
echo 'hey you' | grep -oE '[a-z]+' | while read line;
do
    echo word | wc -c
done >> sometext
```

This is not a construct that you may need to do often, if at all, but hopefully it reinforces pipe usage in Unix.

Less interactive programming

The frequent use of `for` loops, and similar constructs, means that we're moving past the good ol' days of typing in one line of commands and having it execute right after we hit **Enter**. No matter how many commands we pack inside a `for` loop, nothing happens until we hit the `done` keyword.

Write once. Then loop it

With that loss of line-by-line interaction with the shell, we lose the main advantage of the interactive prompt: immediate feedback. And we still have all the disadvantages: if we make a typo earlier in the **block** of commands between `do` and `done`, we have to start all over.

So here's how we mitigate that:

Test your code, one case at a time

One of the biggest mistakes novices make with `for` loops is they think a `for` loop *immediately* solves their problem. So, if what they have to do is download 10,000 URLs, but they can't properly download just *one* URL, they think putting their

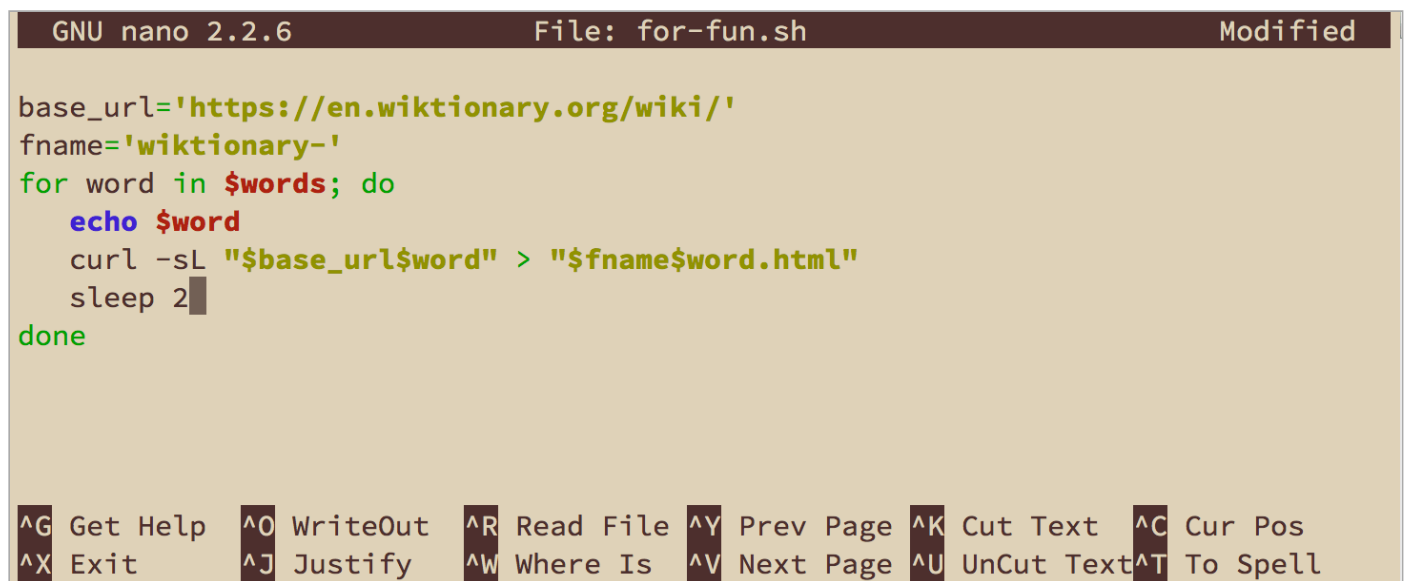
flawed commands into a `for` loop is a step in the right direction.

Besides this being a fundamentally misunderstanding of a `for` loop, the practical problem is that you are now running your broken code *10,000* times, which means you have to wait *10,000* times as long to find out that your code is, alas, still broken.

So pretend you've never heard of `for` loops. Pretend you have to download all 10,000 URLs, one command a time. Can you write the command to do it for the first URL. How about the second? Once you're reasonably confident that no minor syntax errors are tripping you up, *then* it's time to think about how to find a general pattern for the 9,997 other URLs.

Write scripts

The interactive command-line is great. It was fun to start out with, and it'll be fun throughout your computing career. But when you have a big task in front of you, involving more than ten lines of code, then it's time to put that code into a shell script. Don't trust your fallible human fingers to flawlessly retype code.



```
GNU nano 2.2.6           File: for-fun.sh           Modified

base_url='https://en.wiktionary.org/wiki/'
fname='wiktionary-'
for word in $words; do
    echo $word
    curl -sL "$base_url$word" > "$fname$word.html"
    sleep 2
done
```

^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell

Use **nano** to work on loops and [save them as shell scripts \(/topics/bash/scripting\)](/topics/bash/scripting). For longer files, I'll work on my computer's text editor ([Sublime Text \(http://www.sublimetext.com/\)](http://www.sublimetext.com/)) and then upload to the server.

Exercise with web scraping

Just to ground the syntax and workings of the **for-loop**, here's the thought process from turning a routine task into a loop:

For the numbers 1 through 10, use **curl** to download the Wikipedia entry for each number, and save it to a file named "wiki-number-(whatever the number is).html"

The old fashioned way

With just 10 URLs, we *could* set a couple of variables and then copy-and-paste the a **curl** command, 10 times, making changes to each line:

```
user@host:~$ curl http://en.wikipedia.org/wiki/1 > 'wiki-number-1.html'
user@host:~$ curl http://en.wikipedia.org/wiki/2 > 'wiki-number-2.html'
user@host:~$ curl http://en.wikipedia.org/wiki/3 > 'wiki-number-3.html'
user@host:~$ curl http://en.wikipedia.org/wiki/4 > 'wiki-number-4.html'
user@host:~$ curl http://en.wikipedia.org/wiki/5 > 'wiki-number-5.html'
user@host:~$ curl http://en.wikipedia.org/wiki/6 > 'wiki-number-6.html'
user@host:~$ curl http://en.wikipedia.org/wiki/7 > 'wiki-number-7.html'
user@host:~$ curl http://en.wikipedia.org/wiki/8 > 'wiki-number-8.html'
user@host:~$ curl http://en.wikipedia.org/wiki/9 > 'wiki-number-9.html'
user@host:~$ curl http://en.wikipedia.org/wiki/10 > 'wiki-number-10.html'
```

And guess what? It *works*. For 10 URLs, it's not a bad solution, and it's significantly faster than doing it the old *old*-fashioned way (doing it from your web browser)

Reducing repetition

Even without thinking about a loop, we can still reduce repetition using **variables**: the base URL, `http://en.wikipedia.org/wiki/`, and the base-filename never change, so let's assign those values to variables that can be reused:

```
user@host:~$ base_url=http://en.wikipedia.org/wiki
user@host:~$ fname='wiki-number'
user@host:~$ curl "$base_url/1" > "$fname-1"
user@host:~$ curl "$base_url/2" > "$fname-2"
user@host:~$ curl "$base_url/3" > "$fname-3"
user@host:~$ curl "$base_url/4" > "$fname-4"
user@host:~$ curl "$base_url/5" > "$fname-5"
user@host:~$ curl "$base_url/6" > "$fname-6"
user@host:~$ curl "$base_url/7" > "$fname-7"
user@host:~$ curl "$base_url/8" > "$fname-8"
user@host:~$ curl "$base_url/9" > "$fname-9"
user@host:~$ curl "$base_url/10" > "$fname-10"
```

Applying the for-loop

At this point, we've simplified the pattern so far that we can see how little changes with each separate task. After learning about the `for` -loop, we can apply it without much thinking (we also add a **sleep** command so that we pause between web requests)

```
user@host:~$ base_url=http://en.wikipedia.org/wiki
user@host:~$ fname='wiki-number'
user@host:~$ for x in 1 2 3 4 5 6 7 8 9 10
> do
>     curl "$base_url/$x" > "$fname-$x"
>     sleep 2
> done
```

Generating a list

In most situations, creating a for-loop is easy; it's the *creation of the list* that can be the hard work. What if we wanted to collect the pages for numbers 1 through 100? That's a lot of typing.

But if we let our laziness dictate our thinking, we can imagine that counting from x to y seems like an inherently computational task. And it is, and Unix has the `seq` utility for this:

```
user@host:~$ base_url=http://en.wikipedia.org/wiki
user@host:~$ fname='wiki-number'
user@host:~$ for x in $(seq 1 100)
> do
>     curl "$base_url/$x" > "wiki-number-$x"
>     sleep 2
> done
```

Generating a list of non-numbers for iteration

Many repetitive tasks aren't as simple as counting from x to y , and so the problem becomes *how to generate a non-linear list of items?* This is basically what the art of data-collection and management. But let's make a simple scenario for ourselves:

For ten of the 10-letter (or more) words that appear at least once in a headline on the current NYTimes.com front page, fetch the Wiktionary page for that word

We break this task into two parts:

1. Fetch a list of ten 10+-letter words from nytimes.com headlines
2. Pass those words to our for-loop

Step 1: Using the **pup** utility (or command-line HTML parser of your choice):

```
user@host:~$ words=$(curl -s http://www.nytimes.com | \
> pup 'h2.story-heading text{' | \
> grep -oE '[:alpha:]{10,}' | sort | \
> uniq | head -n 10)
```

Step 2 (assuming the `words` variable is being passed along):

```
user@host:~$ base_url='https://en.wiktionary.org/wiki/'
user@host:~$ fname='wiktionary-'
user@host:~$ for word in $words
> do
>     echo $word
>     curl -sL "$base_url$word" > "$fname$word.html"
>     sleep 2
> done
```

Check out [Software Carpentry's excellent guide to for-loops in Bash](http://software-carpentry.org/v5/novice/shell/04-loop.html) (<http://software-carpentry.org/v5/novice/shell/04-loop.html>).

Search

<input type="text" value="Custom Search"/>	<input type="text"/>
--	----------------------

Quick Links

- [Curriculum \(/curriculum\)](#).
- [Homework \(/homework\)](#).
- [Bash guide \(/bash-guide\)](#).
- [Unix tools \(/unix-tools\)](#).
- [Stanford Computational Journalism Lab \(http://cjlabs.stanford.edu\)](http://cjlabs.stanford.edu).
- [Software-Carpentry's guide to the Unix Shell \(http://software-carpentry.org/v5/novice/shell/index.html\)](http://software-carpentry.org/v5/novice/shell/index.html).
- [Data Science at the Command Line \(http://datascienceatthecommandline.com/\)](http://datascienceatthecommandline.com/).

About Computational Methods in the Civic Sphere

[CompCiv \(/\)](#) is a [Stanford Journalism course \(http://journalism.stanford.edu/\)](http://journalism.stanford.edu/) taught by [Dan Nguyen \(http://danwin.com/\)](http://danwin.com/).