

brace expansion

Brace expansion

<http://wiki.bash-hackers.org/syntax/expansion/brace>

```
{string1,string2,...,stringN}
{<START>..<END>}

{<START>..<END>..<INCR>} (Bash 4)

<PREAMBLE>{.....}

{.....}<POSTSCRIPT>

<PREAMBLE>{.....}<POSTSCRIPT>
```

Brace expansion is used to generate arbitrary strings. The specified strings are used to generate **all possible combinations** with the optional surrounding preambles and postscripts.

Usually it's used to generate mass-arguments for a command, that follow a specific naming-scheme.

🔗 It is the very first step in expansion-handling, it's important to understand that. When you use

```
echo {a,b}$PATH
```

then the brace expansion **does not expand the variable** - this is done in a **later step**. Brace expansion just makes it being:

```
echo a$PATH b$PATH
```

Another common pitfall is to assume that a range like {1..200} can be expressed with variables using {a..b}. Due to what I described above, it **simply is not possible**, because it's the very first step in doing expansions. A possible way to achieve this, if you really can't handle this in another way, is using the `eval` command, which basically evaluates a commandline twice:

```
eval echo {a..b}
```

For instance, when embedded inside a for loop :

```
for i in $(eval echo {a..b})
```

This requires that the entire command be properly escaped to avoid unexpected expansions. If the sequence expansion is to be assigned to an array, another method is possible using declaration commands:

```
declare -a 'pics=(img{"$a..$b"}.png)'; mv "${pics[@]}" ../imgs
```

This is significantly safer, but one must still be careful to control the values of `$a` and `$b`. Both the exact quoting, and explicitly including «-a» are important.

The brace expansion is present in two basic forms, **string lists** and **ranges**.

It can be switched on and off under runtime by using the `set` builtin and the option `-B` and `+B` or the long option `braceexpand`. If brace expansion is enabled, the stringlist in `SHELLOPTIONS` contains `braceexpand`.

String lists

```
{string1,string2,...,stringN}
```

Without the optional preamble and postscript strings, the result is just a space-separated list of the given strings:

```
$ echo {I,want,my,money,back}
I want my money back
```

With preamble or postscript strings, the result is a space-separated list of **all possible combinations** of preamble, specified strings and postscript:

```
$ echo _{I,want,my,money,back}
_I _want _my _money _back

$ echo {I,want,my,money,back}_
I_ want_ my_ money_ back_

$ echo _{I,want,my,money,back}-
_I- _want- _my- _money- _back-
```

The brace expansion is only performed, if the given string list is really a **list of strings**, i.e. if there's minimum one `«`, `»` (comma)! Something like `{money}` doesn't expand to something special, it's really only the text `«{money}»`.

Ranges

```
{<START>..<END>}
```

Brace expansion using ranges is written giving the startpoint and the endpoint of the range. This is a «sequence expression». The sequences can be of two types

- integers (optionally zero padded, optionally with a given increment)
- characters

```
$ echo {5..12}
5 6 7 8 9 10 11 12

$ echo {c..k}
c d e f g h i j k
```

When you mix these both types, brace expansion is **not** performed:

```
$ echo {5..k}
{5..k}
```

When you zeropad one of the numbers (or both) in a range, then the generated range is zeropadded, too:

```
$ echo {01..10}
01 02 03 04 05 06 07 08 09 10
```

There's a chapter of Bash 4 brace expansion changes at [the end of this article](#).

Similar to the expansion using stringlists, you can add preamble and postscript strings:

```
$ echo 1.{0..9}
1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9

$ echo ---{A..E}---
---A--- ---B--- ---C--- ---D--- ---E---
```

Combining and nesting

When you combine more brace expansions, you effectively use a brace expansion as preamble or postscript for another one. Let's generate all possible combinations of uppercase letters and digits:

```
$ echo {A..Z}{0..9}
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 C0 C1 C2 C3 C4 C5 C6
C7 C8 C9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 F0 F1 F2 F3
F4 F5 F6 F7 F8 F9 G0 G1 G2 G3 G4 G5 G6 G7 G8 G9 H0 H1 H2 H3 H4 H5 H6 H7 H8 H9 I0
I1 I2 I3 I4 I5 I6 I7 I8 I9 J0 J1 J2 J3 J4 J5 J6 J7 J8 J9 K0 K1 K2 K3 K4 K5 K6 K7
K8 K9 L0 L1 L2 L3 L4 L5 L6 L7 L8 L9 M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 N0 N1 N2 N3 N4
N5 N6 N7 N8 N9 O0 O1 O2 O3 O4 O5 O6 O7 O8 O9 P0 P1 P2 P3 P4 P5 P6 P7 P8 P9 Q0 Q1
Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 S0 S1 S2 S3 S4 S5 S6 S7 S8
S9 T0 T1 T2 T3 T4 T5 T6 T7 T8 T9 U0 U1 U2 U3 U4 U5 U6 U7 U8 U9 V0 V1 V2 V3 V4 V5
V6 V7 V8 V9 W0 W1 W2 W3 W4 W5 W6 W7 W8 W9 X0 X1 X2 X3 X4 X5 X6 X7 X8 X9 Y0 Y1 Y2
Y3 Y4 Y5 Y6 Y7 Y8 Y9 Z0 Z1 Z2 Z3 Z4 Z5 Z6 Z7 Z8 Z9
```

Hey.. that **saves you writing** 260 strings!

Brace expansions can be nested, but too much of it usually makes you losing overview a bit ↗

Here's a sample to generate the alphabet, first the uppercase letters, then the lowercase ones:

```
$ echo {{A..Z},{a..z}}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p
q r s t u v w x y z
```

Common use and examples

Massdownload from the Web

In this example, wget is used to download documentation that is split over several numbered webpages.

wget won't see your braces. It will see **6 different URLs** to download.

```
wget http://docs.example.com/documentation/slides_part{1,2,3,4,5,6}.html
```

Of course it's possible, and even easier, to do that with a sequence:

```
wget http://docs.example.com/documentation/slides_part{1..6}.html
```

Generate a subdirectory structure

Your life is hard? Let's ease it a bit - that's what shells are here for.

```
mkdir /home/bash/test/{foo,bar,baz,cat,dog}
```

Generate numbers with a prefix 001 002 ...

- Using a prefix:

```
for i in 0{1..9} 10; do printf "%s\n" "$i";done
```

If you need to create words with the number embedded, you can use nested brace:

```
printf "%s\n" img{00{1..9},0{10..99},{100..999}}.png
```

- Formatting the numbers with printf:

```
echo $(printf "img%02d.png " {1..99})
```

See the [text below](#) for a new Bash 4 method.

Repeating arguments or words

```
somecommand -v -v -v -v -v
```

Can be written as

```
somecommand -v{,,,}
```

...which is a kind of a hack, but hey, it works.

<div round info>

More fun

The most optimal possible brace expansion to expand n arguments of course consists of n 's prime factors. We can use the «factor» program bundled with GNU coreutils to emit a brace expansion that will expand any number of arguments.

```
function braceify {
    [[ $1 == +([[:digit:]] ) ]] || return
    typeset -a a
    read -ra a < <(factor "$1")
    eval "echo $(printf '{$(printf ,%.s {1..%s})}' "${a[@]:1}")"
}

printf 'eval printf "$arg%s' "$(braceify 1000000)"
```

«Braceify» generates the expansion code itself. In this example we inject that output into a template which displays the most terse brace expansion code that would expand «\$arg» 1,000,000 times if evaluated. In this case, the output is:

```
eval printf  
"$arg"{,,}{,,}{,,}{,,}{,,}{,,}{,,,,}{,,,,}{,,,,}{,,,,}{,,,,}{,,,,}
```

New in Bash 4.0

Zero padded number expansion

Prefix either of the numbers in a numeric range with 0 to pad the expanded numbers with the correct amount of zeros:

```
$ echo {0001..5}
0001 0002 0003 0004 0005
```

Increment

It is now possible to specify an increment using ranges:

$$\{ \langle \text{START} \rangle \dots \langle \text{END} \rangle \dots \langle \text{INCR} \rangle \}$$

<INCR> is numeric, you can use a negative integer but the correct sign is deduced from the order of <START> and <END> anyways.

```
$ echo {1..10..2}
1 3 5 7 9
$ echo {10..1..2}
10 8 6 4 2
```

Interesting feature: The increment specification also works for letter-ranges:

```
$ echo {a..z..3}
a d g j m p s v y
```

See also

- Introduction to expansion and substitution

From: <https://matebcn.eu/> - **miguel angel torres egea**

Permanent link:
<https://matebcn.eu/linux:bash:braceexpansion>

Last update: **24/05/2020 00:37**



