rew

Contents

Overview **2**
How rew works . . . 2

Installation **3**

Usage **3**

Pattern **3**
Syntax . . . 3
Escaping . . . 4

Filters **4**
Path filters . . . 5
Substring filters . . . 8
Field filters . . . 8
Replace filters . . . 9
Regex filters . . . 9
Format filters . . . 10
Generators . . . 11

Input **12**

Output **13**
Diff mode . . . 13
Pretty mode . . . 14

Examples **14**
Path processing . . . 14
Batch rename . . . 14
Batch copy . . . 15
Text processing . . . 15
CSV editing . . . 16

Comparison **16**
rename . . . 16

dirname 16
basename 16
realpath 17
pwd 17
sed 17
cut 17
awk 17
grep 18

Changelog **18**
0.3.0 - 2021-03-29 18
0.2.0 - 2021-02-14 19
0.1.0 - 2020-12-13 20

MIT License **20**

Overview

`rew` is a text processing CLI tool that rewrites FS paths according to a pattern.

How rew works

1. Reads values from standard input.
2. Rewrites them according to a pattern.
3. Prints results to standard output.

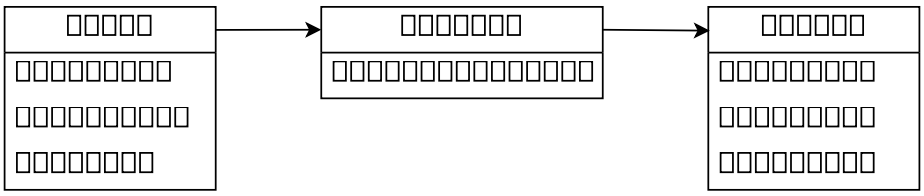

Figure 1: How rew works

Input values are assumed to be FS paths, however, `rew` is able to process any UTF-8 encoded text.

```
find -iname '*.jpeg' | rew 'img_{C}.{e|l|r:e}'
```

`rew` is also distributed with two accompanying utilities (`mvb` and `cpb`) which move/copy files and directories, based on `rew` output.

```
find -iname '*.jpeg' | rew 'img_{C}.{e|l|r:e}' -d | mvb
```

Installation

The latest release is available for download on GitHub.

Alternatively, you can build `rew` from sources:

- Set up a Rust development environment.
- Install the latest release using `cargo`.

  ```
  cargo install rew
  ```

- Binaries will be installed to `.cargo/bin/` in your home directory.

Usage

```
rew [options] [--] [pattern] [values]...
```

When no values are provided, they are read from standard input instead.

```
input | rew [options] [--] [pattern]
```

When no pattern is provided, values are directly copied to standard output.

```
input | rew [options]
```

Use `-d, --diff` flag when piping output to `mvb` / `cpb` utilities to perform bulk move/copy.

```
rew [options] [--] [pattern] -d | mvb
```

Use `-h` flag to print short help, `--help` to print detailed help.

Pattern

Pattern is a string describing how to generate output from an input.

Use `--explain` flag to print detailed explanation what a certain pattern does.

```
rew --explain 'file_{c|<3:0}.{e}'
```

Syntax

By default, pattern characters are directly copied to output.

Input	Pattern	Output
(any)	`abc`	`abc`

Characters `{` and `}` form an expression which is evaluated and replaced in output. Empty expression `{}` evaluates directly to input value.

Input	Pattern	Output
`world`	`{}`	`world`
`world`	`Hello, {}!`	`Hello, world!`

Expression may contain one or more filters, separated by `|`. Filters are consecutively applied on input value.

Input	Pattern	Output	Description
`old.JPEG`	`new.{e}`	`new.JPEG`	Extension
`old.JPEG`	`new.{e|l}`	`new.jpeg`	Extension, Lowercase
`old.JPEG`	`new.{e|l|r:e}`	`new.jpg`	Extension, Lowercase, Remove `e`

Use `-q, --quote` flag to automatically wrap output of every expression in quotes.

```
echo abc | rew {}     # Will print  abc
echo abc | rew {} -q  # Will print 'abc'
echo abc | rew {} -qq # Will print "abc"
```

Escaping

Character `%` starts an escape sequence.

Sequence	Description
`%/`	System directory separator`\` on Windows`/` everywhere else
`%n`	Line feed
`%r`	Carriage return
`%t`	Horizontal tab
`%0`	Null
`%{`	Escaped `{`
`%|`	Escaped `|`
`%}`	Escaped `{`
`%%`	Escaped `%`

Use `--escape` option to set a different escape character.

```
rew '{R:%t: }'              # Replace tabs with spaces
rew '{R:\t: }' --escape='\' # The same thing, different escape character
```

Filters

Filters are categorized into the following groups.

- Path filters
- Substring filters
- Field filters
- Replace filters
- Regex filters
- Format filters
- Generators

Path filters

Path filters assume that their input value is a FS path.

Path components

Filter	Description	Filter	Description
`d`	Parent directory	`D`	Remove last name
`f`	File name	`F`	Last name
`b`	Base name	`B`	Remove extension
`e`	Extension	`E`	Extension with dot

For input value `/home/alice/notes.txt`, filters would evaluate to:

Pattern	Output
`{}`	`/home/alice/notes.txt`
`{d}`, `{D}`	`/home/alice`
`{f}`, `{F}`	`notes.txt`
`{b}`	`notes`
`{B}`	`/home/alice/notes`
`{e}`	`txt`
`{E}`	`.txt`

Parent directory `d` might give a different result than `D` which removes last name of a path. Similarly, file name `f` might not be the same as last name `F` which is a complement of `D`.

Input	`{d}`	`{D}`	`{f}`	`{F}`
`/`	`/`	`/`	*(empty)*	*(empty)*
`/a`	`/`	`/`	`a`	`a`
`a/b`	`a`	`a`	`b`	`b`
`a`	`.`	*(empty)*	`a`	`a`
`.`	`./..`	*(empty)*	*(empty)*	`.`
`..`	`../..`	*(empty)*	*(empty)*	`..`

Input	{d}	{D}	{f}	{F}
(empty)	`..`	*(empty)*	*(empty)*	*(empty)*

Extension with dot `E` can be useful when dealing with files with no extension.

Input	new.{e}	new{E}
old.txt	new.txt	new.txt
old	new.	new

Absolute and relative paths

Filter	Description
`w`	Working directory
`a`	Absolute path
`A`	Relative path

Absolute path `a` and relative path `A` are both resolved against working directory `w`.

{w}	Input	{a}	{A}
/home/alice	/home/bob	/home/bob	../bob
/home/alice	../bob	/home/bob	../bob

By default, working directory `w` is set to your current working directory. You can change that using the `-w, --working-directory` option. `w` filter will always output an absolute path, even if you set a relative one using the `-w` option.

```
rew -w '/home/alice' '{w}' # Absolute path
rew -w '../alice'    '{w}' # Relative to your current working directory
```

Path normalization

Filter	Description
`p`	Normalized path
`P`	Canonical path

Normalized path `p` is constructed using the following rules:

- On Windows, all `/` separators are converted to `\`.

- Consecutive directory separators are collapsed into one.
- Non-root trailing directory separator is removed.
- Unnecessary current directory `.` components are removed.
- Parent directory `..` components are resolved where possible.
- Initial `..` components in an absolute path are dropped.
- Initial `..` components in a relative path are kept.
- Empty path is resolved to `.` (current directory).

Input	Output	Input	Output
(empty)	`.`	`/`	`/`
`.`	`.`	`/.`	`/`
`..`	`..`	`/..`	`/`
`a/`	`a`	`/a/`	`/a`
`a//`	`a`	`/a//`	`/a`
`a/.`	`a`	`/a/.`	`/a`
`a/..`	`.`	`/a/..`	`/`
`./a`	`a`	`/./a`	`/a`
`../a`	`../a`	`/../a`	`/a`
`a//b`	`a/b`	`/a//b`	`/a/b`
`a/./b`	`a/b`	`/a/./b`	`/a/b`
`a/../b`	`b`	`/a/../b`	`/b`

Canonical path `P` works similarly to `p` but has some differences:

- Evaluation will fail for a non-existent path.
- Result will always be an absolute path.
- If path is a symbolic link, it will be resolved.

Directory separator

Filter	Description
`z`	Ensure trailing directory separator
`Z`	Remove trailing directory separator

Directory separator filters `z` and `Z` can be useful when dealing with root and unnormalized paths.

Input	`{}b`	`{}/b`	`{z}b`	`{Z}/b`
`/`	`/b`	`//b`	`/b`	`/b`
`a`	`ab`	`a/b`	`a/b`	`a/b`
`a/`	`a/b`	`a//b`	`a/b`	`a/b`

Substring filters

Filter	Description
`#A-B`	Substring from index `A` to `B`. Indices `A`, `B` start from 1 and are both inclusive. Use `-A` for backward indexing.
`#A+L`	Substring from index `A` of length `L`.
`#A-`	Substring from index `A` to end.
`#A`	Character at index `A`. Equivalent to `#A-A`.

Examples:

Input	Pattern	Output	Input	Pattern	Output
`abcde`	`{#2-3}`	`bc`	`abcde`	`{#-2-3}`	`cd`
`abcde`	`{#2+3}`	`bcd`	`abcde`	`{#-2+3}`	`bcd`
`abcde`	`{#2-}`	`bcde`	`abcde`	`{#-2-}`	`abcd`
`abcde`	`{#2}`	`b`	`abcde`	`{#-2}`	`d`

Field filters

Filter	Description
`&N:S`	Split value using separator `S`, output `N`-th field. Field indices `N` start from 1. Use `-N` for backward indexing. Any other character than `:` can be also used as a delimiter. Use of `/` as a delimiter has special meaning (see below).
`&N/S`	Split value using regular expression `S`, output `N`-th field.
`&N`	Split value using default separator, output `N`-th field.

The default field separator is *horizontal tab*.

- Use `-s, --separator` option to change it to a string.
- Use `-S, --separator-regex` option to change it to a regular expression.

```
echo a1-b2 | rew '{&1} {&2}' -s'-'        # Will print "a1 b2"
echo a1-b2 | rew '{&1} {&2}' -S'[^a-z]+' # Will print "a b"
```

Examples:

Input	Pattern	Output	Input	Pattern	Output
a1\tb2	{&1}	a1	a1\tb2	{&-1}	b2
a1\tb2	{&2}	b2	a1\tb2	{&-2}	a1
a1--b2	{&1:-}	a1	a1--b2	{&-1:-}	b2
a1--b2	{&2:-}	*(empty)*	a1--b2	{&-2:-}	*(empty)*
a1--b2	{&3:-}	b2	a1--b2	{&-3:-}	a1
a1--b2	{&1/[^a-z]+}	a	a1--b2	{&-1/[^a-z]+}	*(empty)*
a1--b2	{&2/[^a-z]+}	b	a1--b2	{&-2/[^a-z]+}	b
a1--b2	{&3/[^a-z]+}	*(empty)*	a1--b2	{&-3/[^a-z]+}	a

Replace filters

Filter	Description
`r:X:Y`	Replace first occurrence of `X` with `Y`. Any other character than `:` can be also used as a delimiter.
`r:X`	Remove first occurrence of `X`. Equivalent to `r:X:`.
`R:X:Y`	Same as `r` but replaces/removes all occurrences.
`R:X`	
`?D`	Replace empty value with `D`.

Examples:

Input	Pattern	Output
ab_ab	{r:ab:xy}	xy_ab
ab_ab	{R:ab:xy}	xy_xy
ab_ab	{r:ab}	_ab
ab_ab	{R:ab}	_
abc	{?def}	abc
(empty)	{?def}	def

Regex filters

Filter	Description
`=E`	Match of a regular expression `E`.
`s:X:Y`	Replace first match of a regular expression `X` with `Y`. `Y` can reference capture groups from `X` using `$0`, `$1`, `$2`, … Any other character than `:` can be also used as a delimiter.
`s:X`	Remove first match of a regular expression `X`. Equivalent to `s:X:`.

Filter	Description
`S:X:Y`	Same as `s` but replaces/removes all matches.
`S:X`	
`@:X1:Y1:...:Xn:Yn:D`	Regular expression switch. Output `Yi` for first `Xi` that matches input. Output `D` when there is no match. `Yi` can reference capture groups from `Xi` using `$0`, `$1`, `$2`, … Any other character than `:` can be also used as a delimiter.
`$0`, `$1`, `$2`, …	Capture group of a global regular expression.

Examples:

Input	Pattern	Output
`12_34`	`{=\d+}`	`12`
`12_34`	`{s:\d+:x}`	`x_34`
`12_34`	`{S:\d+:x}`	`x_x`
`12_34`	`{s:(\d)(\d):$2$1}`	`21_34`
`12_34`	`{S:(\d)(\d):$2$1}`	`21_43`
(any)	`{@:def}`	`def`
`ab`	`{@:^[a-z]+$:lower:^[A-Z]+$:upper:mixed}`	`lower`
`AB`	`{@:^[a-z]+$:lower:^[A-Z]+$:upper:mixed}`	`upper`
`Ab`	`{@:^[a-z]+$:lower:^[A-Z]+$:upper:mixed}`	`mixed`
`a=b`	`{@/(.+)=(.*)/key: $1, value: $2/invalid}`	`key: a, value: b`
`ab`	`{@/(.+)=(.*)/key: $1, value: $2/invalid}`	`invalid`

- Use `-e, --regex` or `-E, --regex-filename` option to define a global regular expression.
- Option `-e, --regex` matches regex against each input value.
- Option `-E, --regex-filename` matches regex against *filename component* of each input value.

```
echo 'a/b.c' | rew -e '([a-z])' '{$1}' # Will print 'a'
echo 'a/b.c' | rew -E '([a-z])' '{$1}' # Will print 'b'
```

Format filters

Filter	Description
`t`	Trim white-spaces from both sides.
`v`	Convert to lowercase.
`^`	Convert to uppercase.
`i`	Convert non-ASCII characters to ASCII.
`I`	Remove non-ASCII characters.
`<<M`	Left pad with mask `M`.

Filter	Description
`<N:M`	Left pad with `N` times repeated mask `M`. Any other character than `:` can be also used as a delimiter.
`>>M`	Right pad with mask `M`.
`>N:M`	Right pad with `N` times repeated mask `M`. Any other character than `:` can be also used as a delimiter.

Examples:

Input	Pattern	Output
`..a..b..`	`{t}`	`a..b` *(dots are white-spaces)*
`aBčĎ`	`{v}`	`abčď`
`aBčĎ`	`{^}`	`ABČĎ`
`aBčĎ`	`{i}`	`aBcD`
`aBčĎ`	`{I}`	`aB`
`abc`	`{<<123456}`	`123abc`
`abc`	`{>>123456}`	`abc456`
`abc`	`{<3:XY}`	`XYXabc`
`abc`	`{>3:XY}`	`abcYXY`

Generators

Unlike other filters, generator output is not produced from its input. However, it is still possible (although meaningless) to pipe input into a generator.

Filter	Description
`*N:V`	Repeat `N` times `V`. Any other character than `:` can be also used as a delimiter.
`c`	Local counter
`C`	Global counter
`uA-B`	Random 64-bit number (`A` $\leq$ `u` $\leq$ `B`)
`uA-`	Random 64-bit number (`A` $\leq$ `u`)
`u`	Random 64-bit number
`U`	Random UUID

Examples:

Pattern	Output
`{*3:ab}`	`ababab`
`{c}`	*(see below)*

Pattern	Output
`{C}`	*(see below)*
`{u0-99}`	*(random number between 0 and 99)*
`{U}`	`5eefc76d-0ca1-4631-8fd0-62eeb401c432` *(random)*

- Global counter `C` is incremented for every input value.
- Local counter `c` is incremented per parent directory (assuming input value is a FS path).
- Both counters start at 1 and are incremented by 1.

Input	Global counter	Local counter
`A/1`	1	1
`A/2`	2	2
`B/1`	3	1
`B/2`	4	2

- Use `-c, --local-counter` option to change local counter configuration.
- Use `-C, --global-counter` option to change global counter configuration.

```
rew -c0   '{c}' # Start from 0, increment by 1
rew -c2:3 '{c}' # Start from 2, increment by 3
```

Input

By default, input values are read as lines from standard input. Each line is expected to be terminated either by `LF` or `CR+LF` characters. The last line (before `EOF`) does not need to have a terminator.

- Use `-t, --read` option to read values terminated by a specific character.
- Use `-z, --read-nul` flag to read values terminated by `NUL` character.
- Use `-r, --read-raw` flag to read whole input into memory as a single value.
- Use `-l, --read-end` flag to read the last value (before `EOF`) only if it is properly terminated.

The following table shows how an input would be parsed for valid combinations of flags/options:

Input	*(no flag)*	`-l`	`-z`	`-lz`	`-t:`	`-lt:`	`-r`
`a\nb`	`a`, `b`	`a`	`a\nb`	*(none)*	`a\nb`	*(none)*	`a\nb`
`a\nb\n`	`a`, `b`	`a`, `b`	`a\nb\n`	*(none)*	`a\nb\n`	*(none)*	`a\nb\n`
`a\0b`	`a\0b`	*(none)*	`a`, `b`	`a`	`a\0b`	*(none)*	`a\0b`
`a\0b\0`	`a\0b\0`	*(none)*	`a`, `b`	`a`, `b`	`a\0b\0`	*(none)*	`a\0b\0`

Input	*(no flag)*	`-l`	`-z`	`-lz`	`-t:`	`-lt:`	`-r`
`a:b`	`a:b`	*(none)*	`a:b`	*(none)*	`a`, `b`	`a`	`a:b`
`a:b:`	`a:b:`	*(none)*	`a:b:`	*(none)*	`a`, `b`	`a`, `b`	`a:b:`

Input values can be also passed as additional arguments. In such case, standard input will not be read.

```
rew '{}' image.jpg *.txt # Wildcard expansion is done by shell
```

Use flag `-I, --no-stdin` to enforce this behaviour even if there are no additional arguments.

```
echo a | rew '{}'       # Will print "a"
echo a | rew '{}' b     # Will print "b"
echo a | rew -I '{}'    # Will print nothing
echo a | rew -I '{}' b  # Will print "b"
```

Output

By default, results are printed as lines to standard output. `LF` character is used as a line terminator.

- Use `-T, --print` option to print results terminated by a specific string.
- Use `-Z, --print-nul` flag to print results terminated by `NUL` character.
- Use `-R, --print-raw` flag to print results without a terminator.
- Use `-L, --no-print-end` flag to disable printing terminator for the last result.

The following table shows how values would be printed for valid combinations of flags/options:

Values	Flags	Output
`a`, `b`, `c`	*(none)*	`a\nb\nc\n`
`a`, `b`, `c`	`-L`	`a\nb\nc`
`a`, `b`, `c`	`-Z`	`a\0b\0c\0`
`a`, `b`, `c`	`-LZ`	`a\0b\0c`
`a`, `b`, `c`	`-T:`	`a:b:c:`
`a`, `b`, `c`	`-LT:`	`a:b:c`
`a`, `b`, `c`	`-R`	`abc`

Apart from this (standard) mode, there are also two other output modes.

Diff mode

- Enabled using `-d, --diff` flag.

- Respects `--print*` flags/options.
- Ignores `--no-print-end` flag.
- Prints machine-readable transformations as results:

```
<input_value_1
>output_value_1
<input_value_2
>output_value_2
...
<input_value_N
>output_value_N
```

Such output can be processed by accompanying `mvb` and `cpb` utilities to perform bulk move/copy.

```
find -name '*.jpeg' | rew -d '{B}.jpg' | mvb # Rename all *.jpeg files to *.jpg
find -name '*.txt'  | rew -d '{}.bak'  | cpb # Make backup copy of each *.txt file
```

Pretty mode

- Enabled using `-p, --pretty` flag.
- Ignores `--print*` flags/options.
- Ignores `--no-print-end` flag.
- Prints human-readable transformations as results:

```
input_value_1 -> output_value_1
input_value_2 -> output_value_2
...
input_value_N -> output_value_N
```

Examples

> Use `rew --explain <pattern>` to print detailed explanation what a certain pattern does.

Path processing

Print contents of the current working directory as absolute paths.

```
rew '{a}' *
```

The previous `*` shell expansion would not work for an empty directory. As a workaround, we can read paths from standard input.

```
dir | rew '{a}'
```

Batch rename

Rename all `*.jpeg` files to `*.jpg`.

```
find -name '*.jpeg' | rew -d '{B}.jpg' | mvb -v
```

The same thing but we generate and execute shell code.

```
find -name '*.jpeg' | rew -q 'mv -v {} {B}.jpg' | sh
```

Normalize base names of files to `file_001`, `file_002`, …

```
find -type f | rew -d '{d}/file_{C|<3:0}{E}' | mvb -v
```

Flatten directory structure `./dir/subdir/` to `./dir_subdir/`.

```
find -mindepth 2 -maxdepth 2 -type d | rew -d '{D}_{F}' | mvb -v
```

Batch copy

Make backup copy of each `*.txt` file with `.txt.bak` extension in the same directory.

```
find -name '*.txt'  | rew -d '{}.bak'  | cpb -v
```

Copy `*.txt` files to the `~/Backup` directory. Preserve directory structure.

```
find -name '*.txt'  | rew -d "$HOME/Backup/{p}"  | cpb -v
```

The same thing but with collapsed output directory structure.

```
find -name '*.txt'  | rew -d "$HOME/Backup/{f}"  | cpb -v
```

The same thing but we also append randomly generated base name suffix to avoid collisions.

```
find -name '*.txt'  | rew -d "$HOME/Backup/{b}_{U}.{e}"  | cpb -v
```

Text processing

Normalize line endings in a file to `LF`

```
rew <input.txt >output.txt # LF is the default output terminator
```

Normalize line endings in a file to `CR+LF`.

```
rew -T$'\r\n' <input.txt >output.txt
```

Replace tabs with 4 spaces.

```
rew '{R:%t:    }' <input.txt >output.txt
```

That would also normalize line endings. To prevent such behaviour, we can process the text as a whole.

```
rew -rR '{R:%t:    }' <input.txt >output.txt
```

Print the first word from each line in lowercase and with removed diacritics (accents).

```
rew '{=\S+|v|i}' <input.txt
```

CSV editing

Swap the first and second column in a CSV file.

```
rew -e'([^,]*),([^,]*),(.*)' '{$2},{$1},{$3}' <input.csv >output.csv
```

The same thing but we use regex replace filter.

```
rew '{s/([^,]*),([^,]*),(.*)/$2,$1,$3}' <input.csv >output.csv
```

Comparison

Let us compare `rew` to a variety of existing tools.

rename

Both `rename` abd `rew` can be used to rename multiple files.

`rename` requires all inputs to be passed as arguments. This means you have to use `xargs` when processing output of `find`. `rew` can read values directly from standard input.

Additionally, `rew` is only a text-processing tool and cannot rename files by itself. You have to use accompanying `mvb` / `cpb` utilities, or you can generate and execute shell code.

```
find -name '*.jpeg' | xargs rename .jpeg .jpg     # Rename *.jpeg files to *.jpg
find -name '*.jpeg' | rew -d '{B}.jpg' | mvb      # The same thing using rew + mvb
find -name '*.jpeg' | rew -q 'mv {} {B}.jpg' | sh # The same thing using rew + mv + sh
```

dirname

Both `dirname` and `rew` can remove last component from a path:

```
dirname 'dir/file.txt'   # Will print "dir"
rew '{d}' 'dir/file.txt' # The same thing using rew
```

basename

Both `basename` and `rew` can remove leading directories from a path:

```
basename 'dir/file.txt'  # Will print "file.txt"
rew '{f}' 'dir/file.txt' # The same thing using rew
```

`basename` can additionally remove filename extension, but we have to manually provide it as a suffix. `rew` is able to remove filename extension automatically:

```
basename 'dir/file.txt' '.txt' # Will print "file"
rew '{b}' 'dir/file.txt'       # The same thing using rew
```

In case the suffix does not represent an extension, `rew` requires an additional filter to remove it:

```
basename 'dir/file_txt' '_txt'   # Will print "file"
rew '{f|s:_txt$}' 'dir/file_txt' # The same thing using rew
```

realpath

Both `realpath` and `rew` can resolve canonical form of a path:

```
realpath -e '/usr/../home' # Will print "/home"
rew '{P}' '/usr/../home'   # The same thing using rew
```

Or they can both compute a relative path:

```
realpath --relative-to='/home' '/usr' # Will print "../usr"
rew -w '/home' '{A}' '/usr'           # The same thing using rew
```

pwd

Both `pwd` and `rew` can print the current working directory:

```
pwd                       # pwd is obviously easier to use
rew '{w}' '<any value>' # rew requires an additional input
```

sed

Both `sed` and `rew` can replace text matching a regular expression:

```
echo '12 ab 34' | sed -E 's/([0-9]+)/_\1_/g' # Will print "_12_ ab _34_"
echo '12 ab 34' | rew '{S:(\d+):_$1_}'       # The same thing using rew
```

cut

Both `cut` and `rew` can print substring:

```
echo 'abcde' | cut -c '2-4' # Will print "bcd"
echo 'abcde' | rew '{#2-4}' # The same thing using rew
```

Or they can both print fields:

```
echo 'ab,cd,ef' | cut -d',' -f2    # Will print "cd"
echo 'ab,cd,ef' | rew -s',' '{&2}' # The same thing using rew
```

awk

`awk` is obviously a more powerful tool than `rew`. However, there are some use cases where `rew` can replace `awk` using more compact pattern syntax.

Printing substring:

```
echo 'abcde' | awk '{print substr($0,2,3)}' # Will print "bcd"
echo 'abcde' | rew '{#2+3}'                # The same thing using rew
```

Printing field:

```
echo 'ab,cd,ef' | awk -F',' '{print $2}' # Will print "cd"
echo 'ab,cd,ef' | rew -s',' '{&2}'       # The same thing using rew
```

Printing first match of a regular expression:

```
echo 'ab 12 cd' | awk 'match($0,/[0-9]+/) {print substr($0,RSTART,RLENGTH)}' # Will print "
echo 'ab 12 cd' | rew '{=\d+}'                                               # The same thi
```

grep

Both `grep` and `rew` can print matches of a regular expression:

```
echo 'ab 12 cd' | grep -Po '\d+' # Will print "12"
echo 'ab 12 cd' | rew '{=\d+}'   # The same thing using rew
```

If an input line contains multiple matches, `grep` will print each on a separate line. `rew` will, however, print only the first match from each line. This is because `rew` transforms lines in 1-to-1 correspondence.

In this particular case, we can workaround it, using raw output mode `-R` and regex replace filters `sS`.

```
echo '12 ab 34' | grep -Po '\d+'                          # Will print "12" and "34"
echo '12 ab 34' | rew -R '{s:^\D+$|S:\D*(\d+)\D*:$1%n}' # The same thing using rew
```

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

0.3.0 - 2021-03-29

Added

- `&` filter which splits value using a separator and outputs N-th column.
- `-q, --quote` flag to automatically wrap output of every pattern expression in quotes.
- `-l, --read-end` flag to require the last input value to be properly terminated.
- `-I, --no-stdin` flag to disable reading values from standard input.

Changed

- `%` is the default pattern escape character instead of `#`.
- `n` filter (substring) was renamed to `#`.
- `N` filter (substring with backward indexing) was replaced by use of `#` with negative indexing (e.g., `#-2`).
- Parsing of `A+L` range can no longer fail with overflow error. Such range would be now resolved as `A-` (from `A` to end).
- Capture groups of a global regex need to be prefixed with `$` (e.g., `{$1}` instead of `{1}`).
- More lenient number parsing that ignore multiple leading zeros (e.g., `001` is interpreted as `1`).
- Output of `--explain` flag and error output have escaped non-printable and other special characters (newline, tab, etc.).
- Output of `--help-pattern` includes list of escape sequences.
- Output of `--help-filters` flag has more readable layout.
- `-T, --no-trailing-delimiter` flag was renamed to `-L, --no-print-end`.
- `-s, --fail-at-end` flag was renamed to `-F, --fail-at-end`.
- `-b, -diff` flag was renamed to `-d, --diff` flag.

Fixed

- `A+L` range is correctly evaluated as "from `A` to `A+L`" (not `A+L+1` as previously).
- `-h, --help` flag displays correct position of `--` argument in usage.

0.2.0 - 2021-02-14

Added

- `@` filter (regular expression switch).
- Alternative way to write range of substring filters as `start+length`.

Changed

- `l` filter (to lowercase) was renamed to `v`.
- `L` filter (to uppercase) was renamed to `^`.
- `0` is now a valid filter a no longer considered error.
- Simplified error message for an invalid range.
- Simplified output of `--help-pattern` and `--help-filters` flags.
- Output of `-h, --help` flag is organized into sections.
- Output of `-h, --help` flag uses more colors in descriptions.
- Regular expression `-e. --regex` / `-E. --regex-filename` is now called *global* instead of *external*.

Fixed

- `--help-filters` flag displays correct name of `i` / `I` filters.

0.1.0 - 2020-12-13

Initial release.

MIT License