Contents

NAME

tren - Advanced File Renaming

HOW TO USE THIS DOCUMENT

tren is a powerful command line file/directory renaming tool. It implements a variety of sophisticated renaming features than can be a bit complex to learn. For this reason, this document is split into two general sections: "REFERENCE" and "TUTORIAL AND DESCRIPTION". If you are new to **tren**, start by studying the latter section first. It will take you from very simple- to highly complex **tren** renaming operations. Once you've got a sense of what **tren** can do, the reference section will be handy to look up options and their arguments.

> **!DANGER!**
>
> **tren** is very powerful and can easily and automatically rename things in ways you didn't intend. It is **strongly** recommended that you try out new **tren** operations with the `-t` option on the command line. This turns on the "test mode" and will show you what the program *would* do without actually doing it. It goes without saying that you should be even more careful when using this program as the system root or administrator. It's quite easy to accidentally rename system files and thereby clobber your OS. You have been warned!!!

REFERENCE

SYNOPSIS

```
tren.py [-aCcdfhqtvXx] [-A alphabet] [-I file] [-i range]  [-P esc] \
        [-R sep] [-r old=new] [-S suffix] [-w width]     file|dir ...
```

SPECIFYING OPTIONS

You may specify *tren* options in one of three ways:

1) On the command line
2) In an "include" file specified with `-I filename` on the command line
3) Via the `$TREN` environment variable

Options specified on the command line are evaluated from left to right and supercede any options specified in the environment variable. Think of any options set in `$TREN` as the "leftmost command line options".

All options must precede the list of files and/or directories being renamed. If one of your rename targets start with the - character, most command shells recognize the double dash as an explicit "end of options" delimiter:

```
tren.py -opt -opt -- -this_file_starts_with_a_dash
```

The list of items targeted for renaming must name directories or files that exist or be a wildcard that expands to existing directories or files. If you specify a non-existent renaming target, `tren` will display an error and stop further processing.

Most shells aren't too fussy about space between an option that takes an argument, and that argument:

```
-i 1
-i1
```

Use whichever form you prefer. Just be aware that there are places where spaces matter. For example, you can quote spaces on your command line to create renaming requests that, say, replace spaces with dashes..

Some options below are "global" - they change the state of the entire program permanently and cannot be undone by subsequent options. Some options are "toggles", they can be turned on- and off as you move from left- to right on the command line. In this way, certain options (like case sensitivity, regular expression handling, and so on) can be set differently for each individual renaming request (`-r`). (If you're very brave, you can select the `-d` option to do a debug dump. Among many other things, the **tren** debugger dumps the state of each renaming request, and what options are in effect for that request.)

OPTIONS

-A alphabet

Install a user-defined "alphabet" to be used by sequence renaming tokens.

(*Default*: Built-in alphabets only)

The alphabet is specified in the form:

```
name:characterset
```

Both the name and the characterset are case- and whitespace-sensitive (if your shell permits passing spaces on the command line). The "0th" element of the alphabet is the leftmost character. The counting base is the length of `characterset`. So, for instance, the following alphabet is named `Foo`, counts in base 5 in the sequence, `a, b, c, d, e, ba, bb, ...`:

```
-A Foo:abcde
```

-a Ask interactively before renaming each selected file or directory.

(*Default*: off)

If you invoke this option, **tren** will prompt you before renaming each file. The default (if you just hit `Enter`) is to *not* rename the file. Otherwise, you have the following options:

```
n - Don't rename the current file

y - Rename the current file

! - Rename all the remaining files
    without further prompting

q - Quit the program
```

These options are all insensitive to case.

If you're doing forced renaming (`-f`), this option will interactively ask you first about making any necessary backups and then renaming the original target. *If you decline to do the backup renaming, but accept the renaming of the original target, the file or directory that already exists with that name will be lost!*.

-b Turn off backups during forced renaming.

(*Default*: Do Backups)

Ordinarily, **tren** will refuse to do a renaming if the new name for a file or directory already exists. You can

override this with the `-f` (forced renaming) option. By default, forced renaming makes a backup copy of the existing file (by appending `.backup` to its name or some other suffix you specify with the `-S` option) before doing the renaming. This prevents existing files from being lost due to a renaming. The `-b` option *inhibits backups* and allows renaming over existing file- and directory names, thereby losing the original file- or directory.

-C Do case sensitive renaming

(*Default*: This is the program default)

This option is provided so you can toggle the program back to its default behavior after a previous `-c` on the command line.

This option is observed both for literal and regular expression-based renaming (`-x`). .

-c Collapse case when doing string substitution.

(*Default*: Search for string to replace is case sensitive)

When looking for a match on the old string to replace, **tren** will ignore the case of the characters found in the file name. For example:

```
tren.py -cr Old=NEW Cold.txt fOlD.txt
```

This renames both files to `CNEW.txt` and `fNEW.txt` respectively. Notice that the new (replacement) string's case is preserved.

This option is observed both for literal and regular expression-based renaming (`-x`).

-d Dump debugging information

(*Default*: Off)

Dumps all manner of information about **tren** internals - of interest only to program developers and maintainers. This

option provides internal program state *at the time it is encountered on the command line*. For maximum debug output, place this as the last (rightmost) option on the command line, right before the list of files and directories to rename. You can also place multiple `-d` options on the command line to see how the internal tables of the program change as various options are parsed.

This option also shows each incremental change to a file name as each renaming request specified on the command line is applied. This can be helful when figuring out a new/complex renaming operation. This is most easily used by invoking the "quiet" and "test" modes:

```
tren.py -tqd -r... -r... file file...
```

-e casetype Force case change to `casetype`.

(*Default*: No forced case.)

This option supports a number of `casetype` arguments to transform the case of the file name:

```
c - Capitalize the file name
l - Force file name to lower-case
s - Swap case of file name characters
t - Force file name to title case
u - Force file name to upper-case
```

"Title case" just means that any alphabetic character following a non-alphabetic character will be capitalized:

```
tren.py -et fee_fi_fo # -> Fee_Fi_Fo
```

Notice that these case transformations are a kind of special built-in renaming request with one important difference: The `-i` "instance" setting is ignored. That's because the `-e` option isn't based on replacing an "old" string like the `-r` renaming option, but rather operates on the file name as a whole.

There is, however, a way to limit the effect of the case forcing options because the `-T` or "target" option *is* observed. You can thus limit the which portion of the file name should have its case changed:

```
tren.py -T4:6 -eu fee_fi_fo # -> fee_FI_fo
```

-f Force renaming even if target file or directory name already exists.

(*Default*: Skip renaming if a file or directory already exists by the same name as the target.)

By default, **tren** will not rename something to a name that is already in use by another file or directory. This option forces the renaming to take place. However, the old file or directory is not lost. It is merely renamed itself first, by appending a suffix to the original file name. (*Default*: .backup, but you can change it via the `-S` option.) This way even forced renames don't clobber existing files or directories.

-h Print help information.

-I file "Include" command line arguments from `file`

It is possible to perform multiple renaming operations in one step using more than one `-r` option on the **tren** command line. However, this can make the command line very long and hard to read. This is especially true if the renaming strings are complex, contain regular expressions or Renaming Tokens, or if you make heavy use of command line toggles.

The `-I` option allows you to place any command line arguments in a separate *file* in place of- or in addition to the **tren** command line and/or the `$TREN` environment variable. This file is read one line at a time and the contents appended to any existing command line. You can even name the files you want renamed in the file, but they must appear as the last lines of

that file (because they must appear last on the command line).

Whitespace is ignored as is anything from a # to the end of a line:

```
# Example replacement string file
# Each line appended sequentially
# to the command line

-xr t[ext]+=txt     # Appended first
-X
-r =/MYEAR/ -r foo=bar
my.file
your.file           # Appended last
```

You may "nest" includes. That is, you can include file x, that includes file y, that includes file z and so on. However, its easy to introduce a "circular reference" when you do this. Suppose file z tried to include file x in this example? You'd be specifying an infinite inclusion loop. To avoid this, **tren** limits the total number of inclusions to 1000. If you exceed this, you'll get an error message and the program will terminate.

Note that wildcard metacharacters like * and ? that are embedded in file names included this way are expanded as they would be from the command shell.

You can define an environment variable, TRENINCL, to specify a path to search to find the named include file(s). tren will use the first (leftmost) instance of the include file it finds along this path. If none are found, tren uses the file path as passed on the command line. This allows relative- and absolute include file paths to be used along with files in an include path:

```
export TRENINCL=$HOME/.tren:/some/where/else
cp file1 $HOME/.tren/
tren.py -I file1 -I foo/file2 -I /foo/bar/file3 *
```

```
'file1' will be found in $HOME/.tren/
'file2' will be found relative to current dir
'file3' will be found on absolute path
```

Note that when defining `TRENINCL`, you must use the path delimiter appropriate for the operating system in question. For Windows, this is `;`, For all other supported OSes, it is `:`.

-i instances

Specifies which "instances" of matching strings should be replaced.

(*Default*: 0 or leftmost)

A file may have multiple instances of the `old` renaming string in it. The `-i` option lets you specify which of these (one, several, all) you'd like to have replaced.

Suppose you have a file called `foo1-foo2-foo3.foo4`. The leftmost `foo` is instance 0. The rightmost `foo` is instance 3. You can also refer to instances relative to the right. So the -1 instance is the last (rightmost), -2, second from the last, and so forth.

Often, you just want to replace a specific instance:

```
-i :3 -r foo=boo
-i :-1 -r foo=boo
```

Both of these refer to the last instance of old string `foo` (found at `foo4` in our example name).

Sometimes, you'd like to replace a whole *range* of instances. An "instance range" is specified using the `:` separator in the form:

```
-i first-to-replace:stop-here
```

Notice that the "stop-here" instance is NOT replaced. In our string above, the option:

```
-i 1:-1 -r foo=boo
```

Would change the file name to:

```
foo1-boo2-boo3.foo4
```

You can also provide partial ranges:

```
-i 1: # Instance 1 to end of name

-i :-2 # Instances to (not including) next-to-last

-i :   # All instances
```

If you provide an instance range that makes no sense or is out of range, `tren` will ignore the argument and leave the instance specification unchanged.

-P char

Use `char` as the escape symbol.

(*Default*: \)

-q

Quiet mode, do not show progress.

(*Default*: Display progress)

Ordinarily, **tren** displays what it is doing as it processes each file. If you prefer to not see this "noisy" output, use the `-q` option. Note that this does not suppress warning and error messages.

It doesn't make much sense to use this option in test mode (`-t`), although you can. The whole point of test mode is to see what would happen. Using the quiet mode suppresses that output.

-R char

Use `char` as the separator symbol in renaming specifications.

(*Default*: =)

-r <old=new>

Replace `old` with `new` in file or directory names.

Use this option to specify which strings you want to replace in each file name. These strings are treated literally unless you also invoke the `-x` option. In that case, `old` is treated as a Python style regular expression.

Both `old` and `new` may optionally contain *renaming tokens* described later in this document.

If you need to use the = symbol *within* either the old or new string, simply escape it: \=

If it is convenient, you can change the separator character to something other than = via the `-R` option. Similarly, you can change the escape character via the `-P` option.

You can have multiple instances of this option on your **tren** command line:

```
tren.py -r old=new -r txt:doc old-old.txt
```

This renames the file to:

```
new-old.doc
```

Remember that, by default, **tren** only replaces the first (leftmost) instance of the old string with the new.

Each rename specification on the command line "remembers" the current state of all the program options and acts accordingly. For example:

```
tren.py -cr A=bb -Cr B=cc ...
```

The `A=bb` replacement would be done without regard to case (both `A` and `a` would match), whereas the `B=cc` request would only replace `B`.

-S suffix

Suffix to append when making backup copies of existing targets.

(*Default*: .backup)

If you choose to force file renaming when the new name already exists (`-f`), **tren** simply renames the existing file or directory by appending a suffix to it. By default, this suffix is `.backup`, but you can change it to any string you like with the `-S '` option.

-T range

Target the range of characters within file name subject to renaming.

(*Default*: Entire file name is subject to renaming.)

Ordinarily, `tren` applies renaming requests and forced case conversions to the entire file name. The `-T` option allows you to specify some substring of the name as the "target" for renaming.

The "range" argument is in the same slice notation used for the `-i` command. So, for example:

```
tren.py -T1:3 -r=XYZ abcdefg # -> aXYZdefg
```

Similarly:

```
tren.py -T 1:4 -es aXYZdefg # -> axyzdefg
```

The `-T` option operates on all subsequent renaming or case forcing operations to the right of it on the command line. So, if you want to go back to the default behavior of applying renaming to the entire file, you have to set the "target" back to the entire file name:

```
tren.py -T1:4 -es -T: -rg=-X axyzdeg # -> aXYZde-
```

As with the `-i` option, range slices that make no sense or are out of range, are simply ignored, and the portion of the file name targeted for renaming is left unchanged.

-t Test mode, don't rename, just show what the program *would* do.

tren is very powerful and capable of doing nasty things to your file and directory names. For this reason, it is helpful to test your **tren** commands before actually using them. With this option enabled, **tren** will print out diagnostic information about what your command *would* do, *without actually doing it*.

If your renaming requests contain random renaming tokens, test mode will only show you an approximation of the renaming to take place (because new random name strings are generated each time the program runs).

-v Print detailed program version information and keep running.

This is handy if you're capturing **tren** output into a log and you want a record

of what version of the program was used.

-w length Set the length of diagnostic and error output.

(*Default*: 80)

tren limits output to this length when dumping debug information, errors, warnings, and general information as it runs. This option is especially useful when you're capturing **tren** output into a log and don't want lines wrapped:

```
tren.py -w999 ..... 2>&1 > tren.log
```

tren makes sure you don't set this to some unreasonably small value such that output formatting would be impossible.

-X Treat the renaming strings literally

(*Default*: This is the program default)

This option is provided so you can toggle the program back to its default behavior after a previous `-x` on the command line.

-x Treat the old string in a `-r` replacement as a Python style regular expression for matching purposes.

(*Default*: Treat the old string as literal text)

TUTORIAL AND DESCRIPTION

> **!DANGER!**
> ONE MORE TIME: **tren** is a powerful file and directory renaming tool. Be **sure** you know what you're about to do. If you're not, run the program in test mode (invoke with the `-t` option) to see what would happen. You have been warned!

The following sections are designed for the new- or occasional **tren** user. They begin with the simplest of **tren** operations and incrementally build more and more complex examples, eventually describing all of **tren**'s capabilities.

Overview

tren is a general purpose file and directory renaming tool. Unlike commands like `mv`, **tren** is particularly well suited for renaming *batches* of files and/or directories with a single command line invocation. **tren** eliminates the tedium of having to script simpler tools to provide higher-level renaming capabilities.

tren is also adept at renaming only *part of an existing file or directory name* either based on a literal string or a regular expression pattern. You can replace any single, group, or all instances of a given string in a file or directory name.

tren implements the idea of a "renaming token". These are special names you can embed in your renaming requests that represent things like the file's original name, its length, date of creation, and so on. There are even renaming tokens that will substitute the content of any environment variable or the results of running a program from a shell back into the new file name.

tren can automatically generate *sequences* of file names based on their dates, lengths, times within a given date, and so on. In fact, sequences can be generated on the basis of any of the file's `stat` information. Sequence "numbers" can be ascending or descending and the count can start at any initial value. Counting can take place in one of several internally defined counting "alphabets" (decimal, hex, octal, alpha, etc.) OR you can define your own counting alphabet. This allows you to create sequences in any base (2 or higher please :) using any symbol set for the count.

A Word About Program Defaults

tren has many options, but its defaults are designed to do two things: a) Simplify the most common operations by making them the default (no options required on the command line), and 2) Reduce the risk of accidentally modifying more of the file name than you intented. So, by default:

> **tren** treats renaming requests *literally*. That is, the "old string" you specify for replacement is treated as literal text. It requires a command line option (`-x`) to treat it as a regular expression. *However*, any renaming tokens found in either the old- or new strings of a renaming request *are* interpreted before the renaming takes place.
>
> **tren** renaming is *case sensitive*. If you want to ignore case, use the `-c` option.
>
> **tren** will only replace the *first (leftmost) instance* of "old string" with "new string". If you want more- or different instances replaced, use the `-i` option.
>
> **tren** will not allow you to rename a file or directory *if one with the new name already exists*. Such attempts will cause no change to the file or directory being processed and an error message will be displayed. This is intentional to force you to manually rename or remove the file or directory that would have been clobbered by a rename. You can override this default and *force* a renaming via the `-f` option. This will cause the orginal file or

directory itself to be renamed with a `.backup` suffix. You can change this suffix via the `-S` option.

Getting Help

There are three command line options that can give you some measure of help and information about using **tren**:

-d	Dumps debug information out to stderr. You can insert multiple instances of this option on the command line to see how the program has parsed everything *to the left* of it. This is primarily intended as a debugging tool for people maintaining **tren** but it does provide considerable information on the internal state of the program that advanced users may find useful.
-h	Prints a summary of the program invocation syntax and all the available options and then exits.
-v	Prints the program version number and keeps running.

Controlling Program Output

As **tren** runs, it produces a variety of diagnostic and status information. There are a number of options you can use to control how this works:

-q	Sets "quiet" mode and suppresses everthing except error messages.
-w num	Tells **tren** to wrap lines after `num` characters have been printed. If you're capturing output to a log, set this to a very high number like 999 to inhibit line wrapping.

Error and debug messages are sent to `stderr`. Normal informational messages are sent to `stdout`. If you want to capture them both in a log, try something like this (depending on your OS and/or shell):

```
tren.py ..... 2>&1 >tren.log
```

Managing Complexity

As you learn more of the program features, the **tren** command line can get long, complex, and easy to goof up. It's also hard to remember all the various options, how they

work exactly, and which specific one you need. For this reason, it is *highly* recommended that - once you have a renaming request working the way you like - if you plan to use it again, save it as an "include" file. That way you can reuse it easily without having to keep track of the details over and over. Instead of this:

```
tren.pu -c -i -1 -r .jpeg=.jpg file ...
```

Do this:

```
tren.py -I jpeg-to-jpg.tren file...
```

What's in the `jpeg-to-jpg.tren` file? Just this:

```
# tren Command Line
# Converts '.jpeg' (in any case mixture) file name suffix to '.jpg'

# Make the replacement case insensitive
-c   # Reset this later on the command line with -C

# Only replace the rightmost instance
 -i -1

# The actual replacement request
-r  .jpeg=.jpg
```

Notice that you can stick comments in the file anywhere you like and that they begin with `#`. Notice also that the various options can be entered on separate lines so it's simpler to read the include file. If you find it useful, you can even include other include files *in* an include file:

```
# Get the jpeg -> jpg suffix renaming

-I  jpeg-to-jpg.tren

# Let's make it fancy

-i -1 -r .jpg=.fancy.jpg
```

If you do this, take care not to create a circular include. This can happen when an include file tries to include itself, either directly, or via another include file. **tren** limits the total number of includes to a very large number. If it sees that the number has been exceeded, it suspects a circular include and will issue an error message to that effect and exit.

You can insert include options anywhere you like on the command line and you can have as many as you like (up to a large number you'll never hit in practice). Each include reference will be replaced with the contents of that file *at the position it appears on the command line*.

If you find yourself using certain options most- or every time you use the program, you can put them in the **$TREN** environment variable. **tren** picks this up every time it starts. This minimizes errors and reduces typing tedium. Just keep in mind that some options can be overriden later on a command line, and some cannot. For instance, suppose you do this:

```
export TREN=-f -c
```

The `-c` option to ignore case can be undone on the command line with a `-C` option. However, the `-f` option cannot be undone.

So ... choose the options you want to make permanent in the environment variable wisely.

Renaming Basics

tren supports a variety of renaming mechanisms. The one thing they have in common is that they're built with one or more *renaming requests* that will be applied to one or more file- or directory names. Renaming requests look like this on the **tren** command line:

```
tren.py ... -r old=new ... -r old=new ... list of files/directories
```

No matter how complicated they look, the basic logic of the renaming request stays the same: "When you find the string `old` in the file- or directory name, change it to the string `new`.

The `old` and `new` renaming strings are built using a variety of building blocks:

Old Strings Are Built With:	*New Strings Are Built With:*
Literal Text	Literal Text
Regular Expressions	Renaming Tokens
Renaming Tokens	

You can use any of these building blocks alone or combine them to create expressive and powerful renaming schemes.

Literal String Substitution

Literal String Substitution is just that - it replaces one literal string with another to rename the target file or directory. This is the most common, and simplest way to use **tren**. This is handy when you have files and directories that have a common set of characters in them you'd like to change. For instance:

```
tren.py -r .Jpeg=.jpg *.Jpeg
```

This would rename all files (or directories) whose names contained the string `.Jpeg` and replace it with `.jpg`. Well ... that's not quite right. Unless you specify otherwise with the `-i` option, *only the first (leftmost) instance of "old" is replaced with "new"*. So, for example, if you started out with the file, `My.Jpeg.Jpeg` and ran the command above, you'd end up with a new file name of `My.jpg.Jpeg`

You can omit either `old` or `new` strings in a renaming specification, but never both.

If you omit the `old` string, you're telling **tren** to *change the whole file name*:

```
tren.py -r =MyNewFilename foo  #New Name: MyNewFilename
```

Be careful with this one. If you apply it to a list of files or directories, it's going to try and name them all to the *same* name. By default, **tren** will refuse to overwrite an existing file name, so it will stop you from doing this. If you absolutely insist on this via the `-f` option, you'll get a bunch of files ending with `.backup`. Say you have files `a`, `b`, and `c`:

```
tren.py -fr =NewName a b c
```

When the command completes, the files will have been renamed in this fashion:

```
a -> NewName.backup.backup
b -> NewName.backup
c -> NewName
```

If you omit the `new` string, you're telling **tren** to *remove* the leftmost instance of `old` string (or other instances via the `-i` option described below) from the file- or directory name. For example:

```
tren.py -rfoo= foo1-foo2-foo3.foo4  # New name: 1-foo2-foo3.foo4
```

If you try to omit *both* `old` and `new` strings, you're effectively telling **tren** to change the existing file name to ... nothing (a null string). This is impossible because file names must be at least one character long. **tren** enforces both this minimum length AND the maximum legal length of new file names. It will print an error and exit if your renaming attempt would violate either of these limits. (As of this writing, the maximum file- or directory name length allowed by the operating systems on which **tren** runs is 255 characters.)

Substitution Instances

As we just saw above, sometimes the `old` string appears in several places in a file- or directory name. By default, **tren** only replaces the first, or leftmost "instance" of an `old` string. However, using the `-i` option you can specify *any* instance you'd like to replace. In fact, you can even specify a *range* of instances to replace.

Instances are nothing more than *numbers* that tell **tren** just where in the name you'd like the replacement to take place. Positive numbers means we're counting instances from the *left* end of the name. The leftmost instance is 0 (not 1!!!).

You can also count *backwards* from the right end of the string using negative numbers. -1 means the last instance, -2 means next-to-last, and so on. In summary, counting from the left starts at zero and counting from the right starts at -1.

Suppose you have a file called:

```
foo1-foo2-foo3.foo4
```

The leftmost `foo1` is instance 0 of old string `foo`. It is also instance -4. The rightmost `foo4` is instance 3 of old string `foo`, and also instance -1.

You can specify a *single instance* to replace:

```
tren.py -i 1 -r f=b foo1-foo2-foo3.foo4 #New Name: foo1-boo2-foo3.foo4
```

```
tren.py -i -1 -r f=b foo1-foo2-foo3.foo4 #New Name: foo1-foo2-foo3.boo4
```

You can also specify a *range of instances* to replace using the notation:

```
-i first-to-replace:stop-here
```

All instances from the "first-to-replace" up to, *but NOT including* "the stop-here" are replaced:

```
tren.py -i 1:3 -r f=b foo1-foo2-foo3.foo4 #New Name: foo1-boo2-boo3.foo4

tren.py -i -4:-2 -r f=b foo1-foo2-foo3.foo4 #New Name: boo1-boo2-foo3.foo4
```

`-i :` means "replace *all* instances":

```
tren.py -i: -r f=b foo1-foo2-foo3.foo4 #New Name: boo1-boo2-boo3.boo4
```

You can also use *partial range specifications*:

```
tren.py -i 1: -r f=b foo1-foo2-foo3.foo4 #New Name: foo1-boo2-boo3.boo4

tren.py -i :-2 -r f=b foo1-foo2-foo3.foo4 #New Name: boo1-boo2-foo3.foo4
```

Note that you cannot specify individual, non-adjacent instances. There is no way to use a single **tren** command to replace, say, the only the 2nd and the 4th instance of an `old` string. Doing that requires two renaming requests. As we'll see in the section below, the good news is that we can do them both on a single **tren** invocation.

Limiting Renaming To Only Part Of Name

Sometimes you just want to rename a *part* of a file- or directory name (aka a name "substring"). As described below, you can use a Regular Expression to do this, but this can be complicated and is often overkill for simple substitutions. **tren** gives you the ability to limit the renaming action to a "targeted" portion of the name using the `-T` option. You simply supply a "slice" describing the portion of the name to be renamed:

```
tren.py -i: -T :3  -r=x  abcdef.text # -> xdef.text
tren.py -i: -T :-4 -re=E abcdef.text # -> abcdEf.text
```

It's important to understand how `-i` and `-T` interact. Even though all instances of a matching old string are specified via the `-i:` option, the `-T` option that follows it limits the portion of the name being considered for renaming. For instance, in the second example, removing the `-T` targeting gives us:

```
tren.py -i: -re=E abcdef.text # -> abcdEf.tExt
```

So, `-T` lets you specify what substring of the full name is a candidate for renaming. All other renaming operations like `-i`, `-r`, and so on operate *only upon the substring specified by* `-T`.

If you specify multiple renaming operations on the command line, any `-T` targeting will remain in effect *for each renaming request*. For instance:

```
tren.py -T -1 -r=OO -r=x foo
```

This will yield incremental renamings of:

```
foo  -> foOO
foOO -> foOx   # Final name
```

In other words, each incremental renaming request honors the current state of the `-T` option. To turn off targeted renaming - that is, make the whole file name the target again - simply include `-T :` on the command line. All renaming requests to the right of it will then target the whole name:

```
tren.py -T -1 -r=OO -T: -rO=x foo  # Yields: foxO
```

More About Slice Notation

Both the `-i` and `-T` options use "slice" notation so it's useful to understand a bit more about how "slices" are constructed.

> **Note**
>
> Although both options use slice notation, they mean very different things. In the case of `-i`, the slice specifies which *instances* of an old string are to be replaced. In the case of `-T`, the slice defines which *characters* in the original file name are "targeted" for renaming.

tren is written in the Python programing language. The slice notation is lifted directly from that language. if you're a Python programmer, you can skip this section :)

Imagine you have a file name like this:

```
abcdef.txt
```

Each character in the name has an "index" or number that tells you what position is occupies in the name. You can count from the *left end* of the name starting with 0:

```
Character         Index From Left
---------         ---------------
a                 0
b                 1
...
x                 8
t                 9
```

You can also count backwards relative to the *right end of the name*:

```
Character         Index From Right
---------         ---------------=
a                 -10
b                 -9
...
x                 -2
t                 -1
```

Notice that left-relative indexes are positive numbers beginning with 0, but right-relative indexes are are negative numbers beginning at -1.

So, what's a slice? A slice is a way of specifying *a range of one or more values*. In the case of the `-T` option, "values" means "positions in the name string targeted for renaming." In the case of the `-i` option, "values" means *which instances of a given string should be renamed*.

In our example above, the `bcd` portion of the name could be defined several different ways:

```
1:4
-9:-6
```

The general form of a slice is:

```
first character/instance:stop on this character/instance
```

This can be tricky to get used to. The number on the righthand side *is not included in he slice* - it is where the slice *ends*.

There are other shortcut forms of slice notation:

```
 :3          # Same as 0:3
3:           # From 4th char/instance through/including end
 :           # All chars/instances are included in the slice
```

In short, slices are a compact way to specify a range of things. If you specify a slice that makes no sense like `-4:3`, **tren** will just ignore it and not do any consequent renaming.

Multiple Substitutions

You can put as many renaming requests on a **tren** command line as you like (.... well, up to the length limit imposed by your operating system and shell, anyway). As we just saw, this can be handy when a single renaming request can't quite do everything we want.

BUT ... there's a catch. In designing your renaming requests, you have to keep in mind that **tren** processes the command line *from left to right*, incrementally constructing the new name as it goes. That is, the leftmost renaming request operates on the original file- or directory name. The next renaming request to the right operates on *that* new name, and so on. In other words, *each renaming request modifies the name produced thus far by all the renaming requests to the left of it on the command line*.

For instance:

```
tren.py -r foo=bar -r foo=baz  foo1-foo2-foo3.foo4
```

Produces ... wait a second ... why on earth are there two renaming requests with identical `old` strings on the same command line? Shouldn't this produce a final name of `baz1-foo2-foo3.foo4`?

Nope. After the leftmost renaming request has been processed, the new name is `bar1-foo2-foo3.foo4`. Remember that, by default, **tren** only replaces the *leftmost* or 0th instance of an `old` string. So, when the second renaming request is processed, the instance 0 of `foo` is now found in the string `foo2`. So, the final name will be, `bar1-baz2-foo3.foo4`.

The lesson to learn from this is that multiple renaming requests on the command line will work fine, but you have to do one of two things (or both):

1) Make sure you're tracking what the "intermediate" names will look like as the new file name is being constructed, renaming request, by renaming request.

2) Make sure the renaming requests operate on completely disjoint parts of the file name.

> **Tip**
> Similarly, **tren** remembers the last state of each option as you move from left to right on the command line. For instance:
>
> `tren.py -i1 -r f=F -r o=O foo1-foo2-foo3.foo4`
>
> You might be tempted to believe that this would produce:
>
> `fOo1-Foo2-foo3.foo4`
>
> But it doesn't. It produces:
>
> `foO1-Foo2-foo3.foo4`
>
> instead because the `-i 1` appears prior to *both* renaming requests and thus applies to each of them. If you want the first instance of "o" to be replaced, you need a command line like this:
>
> `tren.py -i1 -rf=F -i0 -ro=O foo1-foo2-foo3.foo4`
>
> This sort of thing is generally true for *all* options, so be sure they're set the way you want them to the left of a renaming request.

As a practical matter, this can get really complicated to track. If in doubt, it's always better to run two separate **tren** commands in, say, a shell script to make the renaming explicit, rather than to obscure things with clever command line trickery.

So, let's go back to our example from the previous section. We want to replace the 2nd and 4th instances of the string "foo" in our file name. We do this with two renaming requests on the same command line, considering what each one does to the name as it is encountered:

```
tren.py -i1 -r foo=bar -i2 -r foo=bar foo1-foo2-foo3.foo4
```

A good way to get an idea of how incremental renamings *would* take place is to run **tren** is test and debug modes because debug will dump an incremental renaming sequence description as it goes:

```
tren.py -tdq -rfi=fud -et fee_fi_fo
```

The (partial) debug output will show you this:

```
tren.py DEBUG: Renaming Sequence: fee_fi_fo--->fee_fud_fo--->Fee_Fud_Fo
```

More About Command Line Pitfalls

As we just saw, you can get surprising results as **tren** works its way through the command line from left to right. There are other potential pitfalls here, so it's helpful to understand just *how* **tren** processes your command line, step-by-step:

1) Prepend the contents of $TREN to the user-provided command line.

   This allows you to configure your own default set of options so you don't have to type them in every time.

2) Resolve all references to include files.

   This has to be done before anything that follows, because include files add options to the command line.

3) Build a table of every file name to be renamed.

   We'll need this information if any of the renaming requests use the file attribute- or sequence renaming tokens (discussed later in this document).

4) Build a table containing each renaming request storing the current state of every program option at that point on the command line.

   This allows **tren** to apply options differently to different renaming requests on the same command line. This came in handy in our example of the previous section.

5) Resolve any renaming tokens found in either the `old` or `new` portions of the renaming request.

   At this point, both `old` and `new` are nothing more than simple strings (although `old` may be interpreted as a regular expression rather than literally if the option to do so is in effect).

6) Process each file found on the command line in left to right order, applying each renaming request, in the order it appeared from left to right on the command line.

Simple eh? Well, mostly it is ... until it isn't. As we just saw, incrementally building up a new name with multiple renaming requests can produce unexpected results and we have to plan for them.

Similarly, you can inadvertently accidentally give a file the *wrong name entirely* ... this is usually a Bad Thing.

Say you have two files, `x` and `y`. You want to rename `x` to `y` and `y` to `z1`. Well, order matters here. Say you do this:

```
tren.py -fr x=y -r y=z1 x y
```

Let's see what happens in order:

1) File `x` renaming:

   ```
   x -> y
   y -> z1
   ```

   So, file `x` is renamed `z1` (!)

2) File `y` renaming:

```
y -> z1 .... oops, x1 exists, we need a backup

  z1 -> z1.backup

y -> z1
```

Um ... not quite what we wanted. However, if we shuffle around the order of renaming arguments AND the order in which to process the files, we can get what we want:

```
tren.py  -r y=z1  -r x=y   y x
```

Notice that we can drop the `-f` option because there is no longer a naming conflict (see the next section for more about forced renaming).

> **Tip**
> Always remember" **The Rightmost Renaming Request "Wins"!**

The point here, as we've said already, is that you have to be very careful when constructing command lines, keeping track of options, and *what order* you specify both renaming requests *and* the files- and directories to be renamed. As always, the simple way around this is to run multiple, separate **tren** commands, each with its own single renaming request.

Forcing Renaming

By default, **tren** will not allow you to perform a renaming operation if the new name already exists. For example, say you have three files, `a`, `aa`, and `b`, and you try this:

```
tren.py -r a=b a aa b
```

tren will skip the renaming of file `a` because a file named `b` already exists. It will, however, continue to run and rename `aa`, to `ba`.

This is designed to prevent you from accidentally clobbering files that already exist. You can, however, override this default behavior and *force* the renaming to take place in such situations, using the `-f` option. Even then, the existing file isn't lost, it is simply *renamed itself* by appending the suffix `.backup` to its original name. That way, if you made a mistake, you haven't lost the original file. So, in our example above, the command becomes:

```
tren.py -fr a=b a aa b
```

When it's done, we end up with these files:

```
b          # The original 'a' file
b.backup   # The original 'b' file
ba         # The original 'aa' file
```

If you don't like the suffix, `.backup`, you can change it to any string (of length 1 or greater) via the `-S` option:

```
tren.py -S .bku -fr a=b a aa b
```

Now the backed up file will be named `b.bku`.

tren will even backup files that are themselves backups. This can be handy if your renaming request ends up mapping more than one file- or directory name to the same new name:

```
tren.py -fr =newname a b c
```

This produces files named:

```
newname                # The original 'c' file
newname.backup         # The original 'b' file
newname.backup.backup # The original 'a' file
```

You can inhibit this behavior and prevent backups with the `-b` option. This effectively *erases the original file- or directory of that name*. This is **very dangerous** and should rarely be used. It's better to do the backups and delete them later when you're sure you do not need them. The underlying operating system rules for renaming will still apply in this case. For instance, most OSs will not allow you rename a file over the name of an existing directory and vice versa.

> **Note**
>
> The Unix `mv` command will allow you to move a file *into* a directory:
>
> ```
> mv file dir
> ```
>
> However, this is an `mv` "move" semantic, and is not properly a renaming operation. The underlying file system will not permit a file to be renamed over a directory or vice versa. **tren** reflects this OS semantic ... it's not intended to be a reimplementation of `mv`.

Ignoring Case

"Literal" string substitution means just that - **tren** must find an exact instance of `old` in the file name being renamed and replace it with `new`. So, the default is to do *case sensitive* matching. There are times, however, when you want to ignore case when doing this matching. For example, suppose you have file names with a variety of suffixes in various case combinations like `.jpeg`, `.Jpeg`, and `.JPEG`. Suppose you'd like these to all be changed to `.jpg`. Rather than having to do three separate renaming operations it's handy to just ignore case *when matching the old string for replacement*. That's what the `-c` option is for:

```
tren.py -i -1 -c -r.jpeg=.jpg *.jpeg *.Jpeg *.JPEG
```

Notice that the case insensitivity only applies to the *matching* of the `old` string. Once **tren** has determined such a match exists, the `new` string is used *literally* with case intact.

You can turn case sensitivity on- and off for various renaming requests on the same command line. `-C` turns case sensitivity on, and - as we just saw - `-c` turns it off:

```
tren.py -cr X=y -Cr A=b ...
```

The `X=y` renaming request will be done in a case insensitive manner, whereas the `A=b` will be done only on literal instances of upper case `A` in the target file names.

Case Transformation

Sometimes you want to actually force the case of the characters in a filename to change. You do this with the `-e` option. This option takes one of several arguments:

```
c - Capitalize the file name
l - Force file name to lower-case
s - Swap case of file name characters
t - Force file name to title case
u - Force file name to upper-case
```

"Title case" just means that any alphabetic character following a non-alphabetic character will be capitalized:

```
tren.py -et fee_fi_fo # -> Fee_Fi_Fo
```

These case transformations are a kind of special built-in renaming request with one important difference: The `-i` "instance" setting is ignored. That's because the `-e` option isn't based on replacing an "old" string like the `-r` renaming option, but rather operates on the file name as a whole.

You can, however, limit what portion of the filename is "targeted" for case conversion via the `-T` option:

```
tren.py -T 4:6 -et fee_fi_fo # -> fee_Fi_fo
```

As with all renaming requests, `-e` is just another *incremental* renaming operation on the command line:

```
tren.py -rfi=fud -et fee_fi_fo # -> fee_fud_fo -> Fee_Fud_Fo
```

You can actually see these incremental transformations by specifying the `-d` option on the command line.

The Strange Case Of Mac OS X And Windows

Mac OS X and Windows have an "interesting" property that makes case renaming a bit tricky. Both of these operating systems *preserve* case in file and directory names, but they do not *observe* it. (It is possible to change this behavior in OS X when you first prepare a drive, and make the filesystem case sensitive. This is rarely done in practice, however.)

These OSs show upper- and lower- case in file names as you request, but they do not *distinguish* names on the basis of case. For instance, the files `Foo`, `foo`, and `FOO`, are all the same name in these operating systems, and only one of these can exist in a given directory. This can cause **tren** to do the unexpected when your renaming command is doing nothing more than changing case. Suppose you start with a file called `Aa.txt` and run this command:

```
tren.py -rA=a Aa.txt
```

tren will immediately complain and tell you that the file `aa.txt` already exists and it is skipping the renaming. Why? Because from the point-of-view of OS X or Windows, `aa.txt` (your new file name) is the same as `Aa.txt` (your original file name). You can attempt to force the renaming:

```
tren.py -frA=a Aa.txt
```

Guess what happens? Since **tren** thinks the new file name already exists, it backs it up to `aa.txt.backup`. But now, when it goes to rename the original file ... the file is *gone* (thanks to the backup renaming operation)! **tren** declares an error and terminates.

This is not a limitation of **tren** but a consequence of a silly design decision in these two operating systems. As a practical matter, the way to avoid this issue is to never do a renaming operation in OS X or Windows *that only converts case*. Try to include some other change to the file name to keep the distinction between "old name" and "new name" clear to the OS. In the worst case, you'll have to resort to something like:

```
tren.py -rA=X Aa.txt
tren.py -rX=a Xa.txt
```

Using Regular Expressions

Ordinarily **tren** treats both the old string you specify with the `-r` option *literally*. However, it is sometimes handy to be able to write a regular expression to specify what you want replaced. If you specify the `-x` option, **tren** will treat your old string as a regular expresion, compile it (or try to anyway!) and use it to select which strings to replace. This makes it much easier to rename files that have repeated characters or patterns, and groups of files that have similar, but not idential strings in their names you'd like to replace.

Say you have a set of files that are similar, but not identical in name, and you want to rename them all:

```
sbbs-1.txt
sbbbs-2.txt
sbbbbbbbbs-3.txt
```

Suppose you want to rename them, replacing two or more instances of `b` with `X`. It is tedious to have to write a separate literal `-r old=new` string substitution for each instance above. This is where regular expressions can come in handy. When you invoke the `-x` option, **tren** understands this to mean that the `old` portion of the replacement option is to be treated as a *Python style regular expression*. That way, a single string can be used to match many cases:

```
tren.py -x -r bb+=X *.txt
```

This renames the files to:

```
sXs-1.txt
sXs-2.txt
sXs-3.txt
```

Keep in mind that a literal string is a subset of a regular expression. This effectively means that with `-x` processing enabled you can include *both* regular expressions and literal text in your "old string" specification. The only requirement is that the string taken as a whole must be a valid Python regular expression. If it is not, **tren** will display an error message to that effect.

For more detail on the Python regular expression syntax, see:

http://docs.python.org/library/re.html

Because Python regular expressions can make use of the = symbol, you need a way to distinguish between an = used in a regular exression and the same symbol used to separate the old and new operands for the `-r` option. Where this symbol needs to appear in a regular expression, it has to be escaped like this: `\=`. (You can also get around this by changing the `old/new` separator character with the `-R` option.)

As with literal string renaming, regular expression renaming requests honor both the case sensitivity options (`-C` and `-c`) as well as the instance option, `-i`. So, for example:

```
tren.py -x -ci -1 -r Bb+=X sbbsbbbsbbbbsbbbbbs
```

You'll rename the file to `sbbsbbbsbbbbsXs`

Changing The Renaming Separator & Escape Characters

There may be times when the default renaming separator (=) and/or escape character (\) make it clumsy to construct a renaming request. This can happen if, say, either the old- or new string in a literal renaming needs to use the = symbol many times. Another case where this may be helpful is when constructing complex regular expressions that need to make use of these characters.

The `-R` and `-P` options can be used to change the character used for renaming separator and escape character respectively. You can use any character you like (these must be a single character each), but bear in mind that the underlying operating system understands certain characters as being special. Trying to use them here will undoubtedly deeply confuse your command shell, and possibly your file system. For example, the `/` character is used as a path separator in Unix-derived systems. It's therefore a Really Bad Idea to try and use it as a renaming separator or escape character.

Interactive Renaming

By default, **tren** attempts to perform all the renaming requests on all the file- and directory names given on the command line automatically. It is sometimes helpful to work *interactively* wherein you're asked what to do for each proposed renaming. Interactive renaming is requested via the `-a`, "ask" option:

```
tren.py -a -rfoo=Bar foo1.txt foo2.txt foo3.txt
```

tren will compute each file's proposed new name and ask you what you want to do. You have 4 possible choices:

```
N, n, or Enter - No, don't rename this file
Y, y           - Yes, rename the file
!              - Yes, rename everything further without asking
Q, q           - Quit the program
```

There is one slight subtlety here to watch for when doing forced renaming. As we've seen, if you select the `-f` option and the new file name already exists, **tren** will backup the existing file name before doing the renaming. In interactive mode, you will be

asked whether or not to proceed with the renaming both for the file in question *and for any consequenent backups*. If you decline to do the backup but accept the primary renaming, this will have the same effect as the `-b` option: The existing file- or directory will be overwritten by the renaming operation.

If the `-b` option is selected in interactive mode, then you'll only be prompted for the primary file renamings (because `-b` suppresses the creation of backups).

An Overview Of Renaming Tokens

tren implements the notion of *renaming tokens*. These can be a bit complex to grasp at first, so we'll introduce them "gently" in the next few sections and then dive into the detail thereafter.

It is sometimes useful to be able to take a group of files or rename them using some property they possess like creation date, size, owner's name, and so on. This is the purpose of renaming tokens.

Renaming tokens are nothing more than special symbols that represent "canned" information **tren** knows about the file- or directory being renamed, information from the OS itself, and information used to sequence or order the files being renamed.

For instance, if you insert the `/MYEAR/` token into a old- or new string definition, **tren** will replace it with *the year the file or directory being renamed was last modified* and use that string in the renaming process:

```
tren.py -ryear=/MYEAR/ My-year.txt # New name: My-2010.txt
```

Renaming tokens can appear in either the `old` or `new` string components of a `-r` renaming argument. Wherever they appear, they are "resolved" by **tren** before any renaming is attempted. By "resolved", we mean that the renaming token will be *replaced with a string that represents its meaning*. For example:

```
tren.py -i :  -r boo=/SIZE/ boors-and-boots.txt
```

This replaces all the instances of the literal string `boo` with the *length* of the file `boors-and-boots.txt`. When we're done the file will be renamed something like:

```
23rs-and-23ts.txt
```

This is a silly example but it serves to illustrate the point - all renaming tokens get turned into *strings* before any renaming is attempted.

> **Note**
>
> Deep under the covers of it all, **tren** really only knows how to do string replacement. That is, it can replace some `old` string with some `new` string. All the rest of the features you see are sort of syntactic sugar to make it easy for you to express your renaming intent. When **tren** runs, it must resolve all that fancy syntax and boil it down to creating a new file name the underling operating system knows how to produce via its renaming services.

A really handy way to use renaming tokens is to name your files in a particular *order*. For example, suppose you and your friends pool your vacation photos but each of your cameras uses a slightly different naming scheme. You might want to just reorder them by the date and time each picture was taken, for example. That way you end up with one coherent set of named and numbered files. You might start with something like this:

```
DSC002.jpg      # Bob's camera,  taken 1-5-2010 at noon
dc0234.Jpg      # Mary's camera, taken 1-5-2010 at 8am
032344.jpeg     # Sid's camera,  taken 1-3-2010 at 4pm
```

It would be nice to get these in order somehow. We can, by combining *attribute* renaming tokens (that know things about the file being renamed) and *sequence* renaming tokens (that know how to order all the files being renamed by some key like date, length, who owns it, and so on):

```
tren.py -r =/MYEAR//MMON//MDAY/-MyVacation-/+MDATE::0001/.jpeg *.jp*
```

Every place you see something in the form `/.../`, think, "That is a renaming token whose value will be filled in by **tren**." This syntax is the same whether you're using an *attribute-*, *system-*, or *sequence* renaming token.

This would rename all the files in the current directory ending with `.jp*`. The `/MYEAR/...` would be replaced with the *date* the picture was taken (well, actually, the date the file was last modified). The `/+MDATE::0001/` refers to a *starting sequence number* to uniquely identify files modified on the same date. The other strings, `-MyVacation-` and `.jpeg`, are inserted *literally* in the final file names. After we ran this command, the files above would end up with these names:

```
20100103-MyVacation-0001.jpeg       # Sid's
20100105-MyVacation-0001.jpeg       # Mary's
20100105-MyVacation-0002.jpeg       # Bob's
```

Notice that the files taken on the same date have been sequenced by the time-of-day they were taken because we included the `/+MDATE.../` sequence renaming token in our pattern. The + here means to construct the sequence in *ascending* order. A – would specify *descending* order.

> **Note**
>
> Notice that there is *no old string* in our example above. That is, there is nothing to the left of the = symbol in the `-r` option. This effectively means "replace everything" in the existing file or directory name with our newly concocted naming scheme.

Of course, you don't *have* to replace the entire file name when using tokens. It's perfectly legitimate to replace only a portion of the existing name:

```
tren.py -r file=/MYEAR/MMON//MDAY/-file  file-1 file.2
```

This would rename our files to: `20100101-file-1` and `20100101-file.2` Notice that we combined literal text and a renaming token to do this.

You can even use renaming tokens in your *old string* specification. For instance, suppose you manage a number of different systems and you set their system name in an

environment variable called `SYSNAME` and this same name is used to identify backup files. You might then do something like this:

```
tren.py -xr '/$SYSNAME/.*bku$=/FNAME/.old' *
```

If your system name was `matrix`, then the command above would only rename files whose names began with `matrix` and ended with `bku`. If your system name were `morton`, then the command above would only rename files whose names began with `morton` and ended with `bku`.

Notice that we combined a reference to an environment variable within a regular expression. This was done to do the match on "names beginning with... and ending with ...". Also notice that the renaming token `/FNAME/` is just the *original name of the file*.

In order for this to work, we had to single quote the renaming request. This is because Unix shells will themselves try to replace `$SYSNAME` which is not what we want. If we don't single quote (thereby turning off shell variable interpolation) and run this, say, on a machine called "matrix", the command will be handed to **tren** looking like this:

```
tren.py -xr /matrix/.*.bku=/FNAME/.old *
```

tren will then promptly error out and tell you that it doesn't know about a renaming token called `/matrix/`.

There are a several things to keep in mind when doing things like this:

1) The `/$SYSNAME/` in the `old` string is used to *find the text to rename*, whereas the same renaming token in the `new` string means *insert the contents of that environment variable here*.

2) Renaming tokens are always evaluated *before* any regular expression processing takes place. It's up to you to make sure that when the two are combined (as we have in the example above), *that the final result is still a valid Python regular expression*. This may involve explicit quoting of the renaming tokens used in the `old` string specification.

tren has many other kinds of renaming tokens. Their structure and use is described in some detail in the section below entitled "Renaming Tokens: The Gory Details".

Renaming Token Pitfalls

As we saw in earlier sections, **tren** command line option and file name interaction can be tricky. It can depend on order and on whether the various renaming requests "collide" with each other as a new file name is computed. A similar potential collision exists between renaming tokens and renaming requests. Recall from "More About Command Line Pitfalls" that renaming tokens are resolved *before* a renaming request is processed. This means that the string substitution (literal or regular expression) of the renaming operation can *conflict with the characters returned when the renaming token was resolved*. For example, suppose we do this:

```
tren.py -r =New-/FNAME/ -r My=Your MyFile.txt
```

The first renaming request computes the name `New-MyFile.txt`. However, the second renaming request further modifies this to `New-YourFile.txt`. In effect, the

second renaming request is *overwriting part of the string produced by the renaming token reference*. This is an intentional feature of **tren** to allow maximum renaming flexibility. However, you need to understand how it works so you don't get unexpected and strange results. For example, look what happens when you reverse the order of the renaming requests in this case:

```
tren.py -r My=Your -r =New-/FNAME/ MyFile.txt
```

`My` gets replaces with `Your`, but as soon as the second renaming request is processed, the whole string is thrown away and replaced with the final name `New-MyFile.txt`. This is yet another example of, **"The Rightmost Renaming Request Wins"**.

Renaming Tokens: The Gory Details

As we've just seen, a *renaming token* is nothing more than a string representing something **tren** knows about. These fit in one of three categories:

- An attribute of the file or directory being renamed
- An attribute of the underlying operating system environment
- A sequence that reflects some ordering principle

Renaming tokens are delimited by `/` characters, in the form:

```
/RentokenName/
```

tren replaces these tokens with the corresponding information (see descriptions below) wherever you indicated in either the `old` or `new` strings of a `-r` rename command.

Currently, **tren** defines a number of renaming tokens. Future releases of **tren** may add more of these, so it's good to periodically reread this material.

What's The Difference Between An "Attribute" And A "Sequence"?

Some renaming tokens return *attributes* (of either a file or the underling operating system). Some return *sequences*. So, what's the difference?

An "attribute" is a *value* associated with the file- or directory being renamed (or something about the underlying operating system). It could be the length of the file, the last year it was modified, and so on. For example, `/MYEAR/` returns the year the file being renamed was last modified, `/SIZE/` returns the length of the file, and `/FNAME/` returns the original name of the file before renaming. So, if we do this:

```
tren.py -r=/FNAME/-/MYEAR/-/SIZE/ file, file ...
```

Every file will be renamed in the form of:

```
original_name-YYYY-length  # Example: myfile-2010-4099
```

So... *attributes are string substitutions wherein the string tells you something about the file or system on which you're working*.

"Sequences", on the other hand, are just *numbers that represent some ordering principle*. Say you use the sequence renaming token ordered by size, `/+SIZE::001/` to rename 10 files of different sizes:

```
tren.py -r=/+SIZE::01/-/FNAME/ file, file, ...
```

This will produce a new set of files named like this:

```
01-original_name
02-original_name
03-original_name
...
10-original_name
```

Where, `01-original_name` will be the *shortest length* file and `10-original_name` will be the *longest length* file.

So... *sequences are strings of numbers used to put things in some order*.

You can always tell the difference between an attribute- and sequence renaming token, because sequence renaming tokens always start with either a + or − sign (to indicate ascending or descending counting respectively). This distinction is important because some attribute- and sequence renaming tokens share the same name. For instance, `/FNAME/` is an attribute token representing the *original name* of the file before it was renamed. However, `/+FNAME::003/` is a sequence renaming token that returns the *position* (order) of the file name in alphabetic order starting counting from `003`. Although they are both based on the file name (hence the common renaming token symbol), they do very different things.

How tren Uses File Metadata

To keep track of all these attributes and/or to compute sequences, **tren** needs the so-called "metadata" associated with the files- and directories you've named on the command line. This metadata includes information like who owns them, how long they are, what date they were modified, and so on. (This information is commonly described in a data structure called `stat`. Even non-Unix systems like Windows have some version of this data structure.)

The file attribute- and sequence renaming tokens are built on this metadata, so it's worth taking a moment to understand just *how* it is used. **tren** keeps track of the following information for every file- or directory you've named on the command line:

- The order the file appears on *the command line*
- The order the file appears *alphabetically*
- The *original name* of the file before any renaming took place
- The date/time it was last *accessed*
- The date/time it was last *modified*
- The date/time its directory entry (inode) was last *modified*
- The *inode number* for the file

- The *device number* where the directory entry (inode) lives
- The *numeric group ID* the file belongs to
- The *name of the group* the file belongs to
- The *numeric user ID* of the file owner
- The *name of the user* that owns the file
- The *mode or permissions* for the file
- The *number of links* to the file
- The *size* of the file

tren then later uses this information to resolve file attribute renaming tokens, compute the value of a particular sequence renaming token and so on as it finds them in your renaming requests. For example, a sequence renaming token based on group *name* will order the sequence *alphabetically by group name* whereas one based on *group ID* will order it numerically.

It is likely that you'll only be interested in a small subset of these. For completness, though, **tren** keeps track of all the metadata available about the files- or directories named on the command line and makes it available in the form of renaming tokens.

Most commonly, you'll find yourself using the command line, alphabetic, original name, length, and various time/date renaming tokens.

General Attribute Renaming Tokens

These tokens are derived from information about the file or directory being renamed.

> **Note**
>
> **Windows Users Take Note!**
> **tren** is portable across many operating systems because it is written in the Python programming language. Python *mostly* works the exact same way everywhere. However, Windows presents some problems because it does not quite work the same way as Unix-derived OSs do. In particular, if you need to make use of the `/GROUP/` or `/USER/` renaming tokens on Windows, consider installing the `win32all` extensions to your Windows Python installation. If you don't, **tren** will base its order on the generic names `WindowsUser` and `WindowsGroup` which it will apply to every file- or directory under consideration.
> In any case, `/DEV/`, `/GID/`, `/INODE/`, `/NLINK/`, and `/UID/` are not meaningful under Windows and default to 0. Avoid using these tokens on Windows systems, since these will return the same value for every file- or directory.

- `/DEV/` `Returns File- Or Directory's Device ID`

  This is the ID of the device containing the file being renamed. You might want to rename files so that all the files on a given device start

with the same key. That way, their names group together in a sorted directory listing:

```
tren.py -r=/DEV/-/FNAME/ file | dir, file | dir, ...
```

You end up with a sorted directory listing that looks something like:

```
93-...
93-...
97-...
98-...
```

The file names are still preserved in our renaming reqest above, now they're just preceded by the device ID of the where they live with a trailing – separator.

- `/FNAME/  Returns Original File- Or Directory Name`

  This is the name of the file- or directory you are renaming *before* you apply any renaming requests. This allows you to create new names based, in part, on the old name:

```
tren.py -r=/FNAME/-suffix  ...  # Adds "-suffix" to original name
tren.py -r=prefix-/FNAME/  ...  # Adds "-prefix" to original name
tren.py -r /FNAME/=newname ...  # Same as "-r=newname"
tren.py -r /FNAME/=/FNAME/ ...  # Does nothing: old/new are same
```

- `/GID/    Returns File- Or Directory's Group ID`

  This is the number for the group to which the file- or directory belongs. One way to use this is to prepend it to every file name, thereby having all files (and or directories) in the same group list together in a sorted directory listing:

```
tren.py -r=/GID/-/FNAME/ *
```

- `/GROUP/  Returns File- Or Directory's Group Name`

  Essentially the same as `/GID/` except it returns the *name* of the group rather than the number. Again, this is useful when clustering names together in a sorted directory listing:

```
tren.py -r=/GROUP/-/FNAME/ *
```

- `/INODE/  Returns File- Or Directory's Serial Number`

  This is typically an identifier to the directory entry for the file- or directory being renamed. `/DEV/` and `/INODE/` taken together provide a unique systemwide identifier for the file- or directory being renamed.

- `/MODE/   Returns File- Or Directory's Permissions`

  This is a numeric string that represents the permissions of the file- or directory being renamed in standard Unix format.

- `/NLINK/  Returns Number Of Links To File- Or Directory Being Renamed`

Most operating systems allow a single file to have multiple names. These names are "linked" to the instance of the file. This replacement token is a numeric string representing the number of such links.

- `/SIZE/  Returns File- Or Directory's Length In Bytes`

  This is handy if you want a sorted directory listing to list all the files of the same size together. You simply prepend the file- or directory's length onto its name:

  ```
  tren.py -r=/SIZE/-/FNAME/ *
  ```

  Now all of the files of, say, length 23 will group together in a sorted directory listing.

- `/UID/  Returns File- Or Directory's User ID`

  This is the number for the user that owns the file- or directory being renamed. One way to use this is to prepend it to every file name, thereby having all files (and or directories) owned by the same user cluster together in a sorted directory listing:

  ```
  tren.py -r=/UID/-/FNAME/ *
  ```

- `/USER/  Returns File- Or Directory's User Name`

  Essentially the same as `/UID/` except it returns the *name* of the user rather than the number. Again, this is useful when clustering names together in a sorted directory listing:

  ```
  tren.py -r=/USER/-/FNAME/ *
  ```

Time-Related Attribute Renaming Tokens

Modern operating system maintain three different kinds of timestamps for files and directories, `ATIME`, `CTIME`, and `MTIME`:

`ATIME` refers to the last time the file- or directory was *accessed*.

This is updated every time the file is read.

`CTIME` refers to the last time the file- or directory's *inode (directory entry) was modified*.

This is updated whenever a file- or directory's permissions or ownership are changed. It will also be updated when the file- or directory itself is modified.

`MTIME` refers to the last time the file- or directory *itself was modified*.

This is updated whenever the file- or directory is closed after modification.

tren implements a set of time-related file attribute renaming tokens intended to provide full access to these various timestamps. As a practical matter, you're most likely to use the `MTIME`-based tokens, but components for all three time values are available should you need them. They are identically named, except that the first letter of each of the

time-related attribute tokens indicates which of the three timestamps above is used to compute the value:

- `/ADAY/, /CDAY/, /MDAY/` `Returns Timestamp's Day Of The Month`

  Returns the day of the month of the timestamp in `dd` format.

- `/AHOUR, /CHOUR/, /MHOUR/` `Returns Timestamp's Hour Of The Day`

  Returns the hour of the day of the timestamp in `hh` format.

- `/AMIN/, /CMIN/, /MMIN/` `Returns Timestamp's Minute Of The Hour`

  Returns the minute of the hour of the timestamp in `mm` format.

- `/AMON/, /CMON/, /MMON/` `Returns Timestamp's Month Of The Year`

  Returns the month of the year of the timestamp in `mm` format

- `/AMONTH, /CMONTH/, /MMONTH/` `Returns Timestamp's Name Of The Month`

  Returns the name of the month of the timestamp in `Nnn` format.

- `/ASEC/, /CSEC/, /MSEC/` `Returns Timestamp's Seconds Of The Minute`

  Returns the seconds of the minute of the timestamp in `ss` format.

- `/AWDAY, /CWDAY/, /MWDAY/` `Returns Timestamp's Name Of The Weekday`

  Returns the name of the day of the timestamp in `Ddd` format.

- `/AYEAR, /CYEAR/, /MYEAR/` `Returns Timestamp's Year`

  Returns the year of the timestamp in `yyyy` format.

So, for example:

```
tren.py -r=/FNAME/-/MYEAR/-/MMON/-/MDAY/-/MMONTH/-/MWDAY/-/MHOUR/:/MMIN/:/M
```

Might rename the file to something like:

```
foo-2005-01-07-Jan-Fri-01:23:33
```

System Renaming Tokens

These tokens are derived from the underlying operating system and runtime environment. Notice that, because command interpreters (shells) on various systems work differently, the first two of these have to be quoted in different ways.

- `/NAMESOFAR/ Current state of new name`

`tren` allows multiple renaming requests to be specified on the command line . Each of these operates serially on the renaming target name: The leftmost request operates on the original name. The resulting name is handed to the next request to the right and so on.

`/NAMESOFAR/` allows the current state of a new name to be included explicitly in a renaming request. i.e., You can insert the name a renaming request starts out with into its own renaming specification:

```
tren.py -rX=y -r=/NAMESOFAR/.text  Xray.txt
```

The first renaming request transforms the name from `Xray.txt` to `yray.txt`. This is thus the "name so far" with which the second request begins. So, the second renaming request transforms the name `yray.txt` into `yray.txt.text`.

- `/$ENV/` Environment variable

  This token is replaced with the value of the environment variable `ENV`. If that variable does not exist, the token is replaced with an empty string:

  ```
  tren.py -r ='/$ORGANIZATION/'-/FNAME/ *  # Unix shells
  tren.py -r =/$ORGANIZATION/-/FNAME/ *    # Windows shells
  ```

  This prepends the organization's name to everything in the current directory.

- /`cmd`/ Arbitrary command execution

  This token is replaced with the string returned by executing the `cmd` command. Note that newlines are stripped from the results, since they don't belong in file names. Spaces, however, are preserved.

  For instance, you might want to prepend the name of the system to all your shell scripts:

  ```
  tren.py -r ='/`uname -n`/'-/FNAME/ *.sh  # Unix shells
  tren.py -r ="/`uname -n`/"-/FNAME/ *.sh  # Windows shells
  ```

  This construct is more generally a way to synthesize renaming tokens that are not built into **tren**. You can write a script to do most anything you like, execute it within the /`cmd`/ construct, and plug the results into your new file name. This effectively provides **tren** an unlimited number of renaming tokens.

Warning

Be *very* careful using this. It's possible to construct bizzarre, overly long, and just plain chowder-headed strings that make no sense in a file name using this token. Moreover, if you attempt to insert characters that don't belong in a file- or directory name (like a path separator), construct a file name that is too long (or too short), or just generally violate something about the filesystem's naming rules, this will cause **tren** to abort and spit out an error. *However*, you will not be prevented from creating file names that are legal but undesirable, such as file names that begin with the – character. In other words, be careful and be sure you know what you're doing with this renaming token.

Tip

MORE ABOUT QUOTING `/$ENV/` **AND** `/`cmd`/` **SYSTEM RENAMING TOKENS**

Both of these constructs are supported directly from most Unix command shells. That is, most Unix shells will themselves dereference constructs like `$ENV` and `` `command` ``. There's no need to pass them as renaming tokens, you can just use the shell's capabilities:

```
tren.py -r =/FNAME/-`uname -n`-$LOGNAME
```

If you do want to use the renaming token form in a Unix shell, you *must* single quote them to prevent the shell from "interpolating" the variables before **tren** is called. If you don't do this, **tren** will complain about encountering unknown renaming tokens:

```
tren.py -r='/`uname -n`/'-/FNAME/ *.sh # Right
tren.py -r=/`uname -n`/-/FNAME/ *.sh   # Wrong
```

The real reason for providing these renaming tokens at all is because the Windows command interpreter does not have an equivalent function. The *only* way to achieve what these do on Windows is via renaming tokens. In Windows, you also have to pay attention to quoting, particularly when there are spaces in a `` `cmd` `` renaming token:

```
tren.py -r=/FNAME/-/`command opts args`/ ...
```

This causes **tren** to complain mightily because it thinks `` /`command ``, `opts`, `args`, are all separate (invalid) command line arguments. To avoid this problem, you need to pass the renaming token as a single command line entity via quotes:

```
tren.py -r=/FNAME/-"/`command opts args`/"  ...
```

- `/RAND#/` Random Number Generator

  This generates a (quasi) random number string, `#` digits wide.

  This can be useful when you want to guarantee that no renaming operation will generate a new name that conflicts with an existing name:

  ```
  tren.py -r=/MYEAR//MMON//MDAY/-/RAND10/ *
  ```

  This generates new file names with a 10 character random number string suffix:

  ```
  20100401-4708910871
  ```

In this case, just make sure the random number string is long enough to make a name collision unlikely by picking a sufficiently large #.

must be a positive integer greater than 0. The random number generator is reinitialized each time the program runs, so test mode operations will only show you the "shape" of the names with the embedded random number strings, not the actual strings you'll end up with.

Another nice use of this feature is to "mask" the actual file names. Say you have a bunch of encrypted files, but you don't want a casual viewer to even know what they are or what's in them. You might do this:

```
tren.py -r=/RAND25/ * 2>&1 >tren.log
```

Now you can encrypt `tren.log` and send it along with the files themselves over a non-secure channel. The recipient can decrypt the log, and figure out what the original file names were, decrypt them, and store them accordingly.

Sequence Renaming Tokens

Sometimes it's useful to rename files or directories based on some *property they possess* like the date or time of creation, the size of the file, who owns it, and so on. That's the idea behind the attribute renaming tokens described in the previous sections.

But another really interesting use of renaming tokens is to *order all the files being renamed* based on one of these parameters. For instance, instead of actually embedding the date and time of creation in a file or directory name, you might want to order the files from oldest to newest with a naming convention like:

```
file-1.txt
file-2.txt
file-3.txt
```

This guarantees uniqueness in the final name and also sees to it that a sorted directory listing will show you the files or directories in the order you care about.

This is the purpose of *sequence renaming tokens*. They give you various ways to create sequences that can be embedded in the final file or directory name.

> **Tip**
> Many sequence renaming tokens described below share the same name with an attribute renaming token described in the previous sections. That's because they are based on the same property of the file- or directory being renamed. However, it's easy to tell which is which: Sequence renaming tokens always begin with either + or − (to indicate ascending- and descending ordering respectively).
> So, `/GROUP/` is an attribute renaming token that returns the group *name* for the file. However, `/+GROUP.../` is a sequence renaming token that returns a number indicating what *position* the file is in when all the files named on the command line are *ordered by their group names*.

General Format Of Sequence Renaming Tokens

Sequence renaming tokens consist of four descriptive components and have the following general format:

```
/OrderingType:Counting Alphabet:Counting Pattern/

  where,
         Ordering (Required):

               +  ascending
               -  descending

         Type (Required):

               The attribute used to create the ordering.

         Counting Alphabet (Optional):

               The name of the counting system to use.

         Counting Pattern (Optional):

               Establishes the first value in the counting
               sequence and/or provides a string to format
               the count.
```

Note that there is no space between the *Ordering* flag and *Type*.

An *Ordering* flag is mandatory. It will either be + to indicate an ascending count or – to indicate a descending count.

The *Type* is mandatory. These are documented in the section below entitled, "Types Of Sequence Renaming Tokens".

The *Counting Alphabet* is optional. Counting alphabets are ways to count in different bases and even to use something other than just numbers to represent the count. These are described in the section below entitled, "Let's Learn The Alphabet".

If you omit naming a specific alphabet, **tren** will default to counting in Decimal. Note that you *cannot* omit the alphabet delimiters, so the correct form of a sequence renaming token then becomes:

```
/OrderingType::Counting Pattern/
```

A *Counting Pattern* is optional. Counting patterns are used to do two things: Set the initial value for the count and Describe the layout of how the count should look. This is described in the section below entitled, "Counting Pattern Format".

If you omit a counting pattern, **tren** will start counting from the zero-th "number" in your chosen alphabet, producing a counting pattern as "wide" as necessary to count all the items being renamed. In that case, the format of a sequence renaming token becomes:

```
/OrderingType:Alphabet:/    # With explicit alphabet
/OrderingType::/            # With default decimal alphabet
```

Let's Learn The Alphabet

Sequence renaming tokens are essentially "counters" that return a number string representing where the file- or directory being renamed sits in some order - say, by time, alphabetically or on the command line.

To be as flexible as possible in creating renaming strings, it's helpful to be able to "count" in any *base*, and use any *set of symbols* when counting. For instance, you may prefer sequences of letters instead of numbers. Such a sequence might look like this:

```
 a
 b
 ...
 z
aa
ab
...
az
ba
bb
```

And so on.

tren has a number of standard such "counting alphabets" built in for the most common counting situations. As described in the previous section, you specify which of these you want to use in each sequence renaming token reference on the command line. (If you omit naming a specific alphabet, the token will default to counting in Decimal.)

The built in alphabets are:

```
Binary     -  Counting in Base 2 using numbers
Octal      -  Counting in Base 8 using numbers
Decimal    -  Counting in Base 10 using numbers
HexLower   -  Counting in Base 16 using numbers and lower case letters
HexUpper   -  Counting in Base 16 using numbers and upper case letters
Lower      -  Counting in Base 26 using lower case letters
LowerUpper -  Counting in Base 52 using lower- then upper case letters
Upper      -  Counting in Base 26 using upper case letters
UpperLower -  Counting in Base 52 using upper- then lower case letters
```

Tip

The difference between a "base" and a "symbol set".
In order to make such counting-based renamings as flexible as possible, **tren** is built to be able to count *in any base* (2 or higher) and *make use of any symbol set*. What's the difference? The "base" tells you how many symbols there are in your counting system. In Decimal, for example, there are 10. The "symbol" set, assigns a character to represent each of those positions. In Decimal, we customarily use, "0", "1", "2", and so on. However, there is nothing magical about the symbol set. It is the *base* that defines the counting system. The symbol set is just an arbitarary representation. For instance, there's no reason we can count in base 10, using the symbols, ")", "!", "#", "$", ... and so on.

This ability to use any symbol set in any base makes it easy to construct counting strings that suit your particular renaming needs. You do this by defining your own, custom counting "alphabet" via the `-A` command line option:

```
-A AlphabetName:string-of-characters
```

Once defined, later renaming tokens on the command line can refer to it via the `/...:AlphabetName:.../` syntax discussed previously.

Say we do this:

```
tren.py -A Foo:s2X -r=/+MTIME:Foo:/ *
```

This will rename all the files in the current directory in ascending `mtime` timestamp order using the following counting scheme:

```
 s
 2
 X
ss
s2
sX
2s
22
2X
```

And so on. You can use most any combination of characters you like to customize your sequence renaming token output. There are a few things to keep in mind, however:

- The counting *base* is determined by the *number of symbols in the symbol set* not what characters you use. In the example above, we're counting in base 3 irrespective of what symbols are used to represent each "number".
- You can define as many new alphabets as you like on the command line. (Well ... up to the maximum command line length limit imposed by the shell and/or operating system you're using.)
- The alphabet name is case sensitive. `Foo`, `FOO`, and `foo` are all different alphabet names (assuming they are all defined).
- There is no requirement that the symbol set be built out of unique characters. **tren** does no analysis of your symbol set at all, so this is permitted (if not recommended):

```
-A  Foo:abcx123xj3,m2
```

- Similarly, you can populate your alphabet with any symbols you like, BUT remember they're going to be embedded in some file- or directory name. It's a good idea to make sure you avoid illegal or undesirable characters like `/`, `\`, and `-` in your alphabets so they don't end up getting embedded in a name (or trying to, anyway).
- If you use non-numerical counting schemes, your sorted directory list will not reflect that order. For example, suppose you have a bunch of files in a directory, and you do this:

  ```
  tren.py -r=/-MTIME:LowerUpper:/ *
  ```

  Your files will get renamed in descending `mtime` timestamp order as:

  ```
  a
  b
  ...
  A
  B
  ...
  aa
  ```

  And so on, where `a` is the oldest file- or directory. However, when you do a sorted directory listing, the names beginning with *upper case* characters will be listed first. Why? Because directory sorting is typically based on ASCII order wherein `A-Z` appear before `a-Z`.

Counting Pattern Format

When using sequence renaming tokens, it's nice to be able to layout the resulting counting string in some consistent way. You can use an optional "counting pattern" in your sequence renaming token to do this. The renaming pattern is used to specify two things: The width of the sequence string, and, optionally, the starting value for the sequence. For instance:

```
Pattern      Results
-------      -------

0001    ->   0001, 0002, 0003, ...
0000    ->   0000, 0001, 0002, ...
03      ->   03, 04, 05, ...
```

To understand counting patterns, you have to understand a few basic rules **tren** uses to interpret them:

- The number of characters (of any kind) in the pattern fix the *width* of the counting string. These characters need not even be in the counting alphabet:

  ```
  tren.py -r=/+CTIME::abcde/ *
  ```

This produces files renamed in ascending `ctime` timestamp order like this:

```
abcd0
abcd1
...
abc10
```

And so on.

- When a count increments such that it would exceed the width of the pattern, it "rolls over" and **tren** issues a warning message to that effect. Using the example above, we'd get:

```
9998
9999
0000  # Count rolls over and warning issued!
```

Notice that the count rolls over *in the selected counting alphabet*, it does not restart from the original counting pattern. In almost every case, you should avoid roll over and make your counting pattern wide enough to hold a full count for all the files- and directories you've named on the command line. One issue here is that rolling over is possibly going to create a name collision and the renaming will either be skipped or have to be forced (with backup) using the `-f` option.

- As we've seen, **tren** treats each position of the counting pattern as a placeholder and "eats" characters as the count goes up. This allows you great flexibility in creating renaming patterns that embed both a count and *a literal* string via a single sequence renaming token. You just have to make the counting pattern wide enough so that the highest count never consumes your literal string:

```
tren.py -r=/+MTIME:HexLower:InHexMtimeOrder-0x00000/ *
```

This yields new file names like this:

```
InHexMtimeOrder-0x00000
InHexMtimeOrder-0x00001
InHexMtimeOrder-0x00002
...
```

Notice that the `0x` string may mean "this is a hex number" to the human reader, but it is completely insignificant to **tren**. If the count were to get large enough - bigger than 5 digits, the `0x` string itself would get overwritten. Larger still, and `InHexMtimeOrder-` would start to get consumed.

Tip

We could avoid the possibility of having the count ever consume our literal text, by taking it *out of the sequence renaming token* and putting it in as a literal argument to the `-r` option, thereby separating the text from the count:

```
-r=InHexMtimeOrder-0x/+MTIME:HexLower:00000/
```

In short, **tren** treats *every character in a counting pattern the same* - with complete indifference.

- Well ... *almost* "complete indifference". When **tren** finds characters that are *in* the selected counting alphabet, it *adds them to the count*. In this way we start counting at some predermined initial value. Note that **tren** always produces sequence number *starting with 0* and, unless the pattern indicates otherwise:

```
tren.py -r=/+CMDLINE::/ a b c
```

Produces:

```
0   # Formerly a, the 1st command line argument
1   # Formerly b, the 2nd command line argument
2   # Formerly c, the 3nd command line argument
```

But say we wanted to start counting from 1 instead:

```
  tren.py -r=/+CMDLINE::1/ a b c

Produces::

  1   # Formerly a, the 1st command line argument
  2   # Formerly b, the 2nd command line argument
  3   # Formerly c, the 3nd command line argument
```

Similarly, `/+CMDLINE::101/` would produce:

```
101 # Formerly a, the 1st command line argument
102 # Formerly b, the 2nd command line argument
103 # Formerly c, the 3nd command line argument
```

Because **tren** is insensitive to characters *outside* the counting alphabet, you can produce really interesting counting patterns like this:

```
  tren.py -r=/+CMDLINE::1x0/ a b c

Produces::

  1x0   # Formerly a, the 1st command line argument
  1x1   # Formerly b, the 2nd command line argument
  1x2   # Formerly c, the 3nd command line argument

If you had enough files named on the command line, the
count would eventually consume the out-of-alphabet
characters::

  1x0
  ...
  1x9
  110
```

```
111
...
```

So, by mixing characters that are both in- and out of the counting alphabet in a counting pattern, you "prime" the sequence renaming token to start counting with a certain string. Notice that you can do this in *any* position within the pattern. Say you do this:

```
tren.py -r=/+CMDLINE::x1x4/ *
```

This will produce a counting sequence like this:

```
x1x4
x1x5
...
x110
...
x200
```

In other words, a character in any position of the pattern that is in the counting alphabet will be added to the count.

This works for all alphabets, any base, and any symbol set:

```
tren.py -r=/+FNAME/:Upper:+0S/ *
```

Yields new file names:

```
+0S
+0T
...
+0Z
+BA
+BB
...
```

- There is no notion of starting the count from a "negative number" and counting up. You can sort of synthesize this by sticking a – in front of a sequence renaming token (or at the left end of its counting pattern). Keep in mind, though, that **tren** only knows how to *increment* a count so you will always get an "increasing negative number" when you do this:

  ```
  tren.py -r=-/+CMDLINE::5/-/FNAME/ a b c
  ```

  Will produce new file names:

  ```
  -5-a
  -6-b
  -7-c
  ```

  If you want the reverse order, specify a descending sequence renaming token:

  ```
  tren.py -r=-/-CMDLINE::5/-/FNAME/ a b c
  ```

  Will produce new file names:

```
-5-c
-6-b
-7-a
```

Types Of Sequence Renaming Tokens

Sequence renaming tokens are thus a way to generate an ordering *based on some property common to everything being renamed*. That property is used to return a string representing just where in that order a particular file- or directory appears. This string is formatted according to the counting alphabet and counting pattern embedded in the sequence renaming token as described in the previous sections.

Keep in mind that for purposes of sequencing, **tren** *makes no distinction between a file and directory*. It merely sequences based on the property you requested.

> **Note**
>
> There is one *very* important detail to keep in mind here. When **tren** first starts up, it examines the metadata of every file- and directory name on the command line. It uses this to pre-create the sequences for every possible ordering (alphabetic, by date, within date, by length, and so on) *whether or not every file actually ends up being renamed later on*. In other words, sequences are built on *the list of names passed on the command line* NOT on the list of files- or directories that actually get renamed. If your renaming requests only apply to some of the file names you passed on the command line, you may find the resulting sequence unexpected. Say you have three files, `a`, `b`, and `c` and you do this:
>
> ```
> tren.py -rb=/FNAME/-/+FNAME::001/ b c a
> ```
>
> Only file `b` has a matching `old` string and thus is the only file renamed. However, because it is second alphabetically *of all the files named on the command line*, it gets renamed to `b-002`. The way to avoid this surprise is to make sure any renaming request with sequence renaming tokens in it is constructed so that it applies to *all* the files- and directories named on the command line.

Sometimes, more than one file- or directory named on the command line maps to the same sequencing key. For example, when using the `/+GROUP.../` sequence renaming token, dozens of files in a given directory may only map to a few group names. In this situation, all the names that map to the same key *will be sequenced alphabetically within the key*. So if `a` and `b` are in group `foo` and `c` and `d` are in group `baz`:

```
tren.py -r=/+GROUP/::/-/FNAME/ a b c d
```

Will create the new names:

```
0-c
1-d
2-a
3-b
```

tren currently supports a variety of sequence renaming tokens. Note that those associated with the various OS timestamps begin with the corresponding first letter:

- `/+-ADATE:Alphabet:Pattern/` Sequence based on atime WITHIN the same date
- `/+-CDATE:Alphabet:Pattern/` Sequence based on ctime WITHIN the same date
- `/+-MDATE:Alphabet:Pattern/` Sequence based on mtime WITHIN the same date

  These return sequences *within* a given day. This enables renaming constructs like:

  ```
  tren.py -r=/MYEAR//MMON/MDAY/-/+MDATE::001/ *
  ```

  Yielding files named:

  ```
  20100305-001
  20100305-002
  20100305-003
  20100316-001
  20100316-002
  20100316-003
  ...
  ```

- `/+-ATIME:Alphabet:Pattern/` Sequence based on atime timestamp
- `/+-CTIME:Alphabet:Pattern/` Sequence based on ctime timestamp
- `/+-MTIME:Alphabet:Pattern/` Sequence based on mtime timestamp

  These return sequences in absolute timestamp order. For example:

  ```
  touch foo
  touch bar
  touch baz
  tren.py -r =/+MTIME::/-/FNAME
  ```

  Yields:

  ```
  0-foo
  1-bar
  2-baz
  ```

- `/+-CMDLINE:Alphabet:Pattern/` Sequence based on the order of appearance on the command line

  This is nothing more than the command line order:

  ```
  tren.py -r=/+CMDLINE/-/FNAME::01/-/FNAME/ z b a
  ```

  Yields:

  ```
  01-z
  02-b
  03-a
  ```

- `/+-DEV:Alphabet:Pattern/` Sequence based on the device ID number on which the file- or directory resides

This is the a sequence ordered by which device ID contains the file- or directory to be renamed.

This is not supported on Windows and defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

- `/+-FNAME:Alphabet:Pattern/  Sequence based on alphabetic order of all targets on the command line`

  This returns a sequence based on the alphabetic order of everything you've named for renaming. Note that this is done on *the fully qualified path name for each argument*, not just the file- or directory name itself:

  ```
  tren.py -r=/+FNAME::/-/FNAME/ a/z b/b
  ```

  Yields:

  ```
  a/0-z
  b/1-b
  ```

  This is because the original file name `a/z` sorts alphabetically before `b/b`.

- `/+-GID:Alphabet:Pattern/  Sequence based on the group ID number`

  This returns a sequence ordered by the ID number of the group to which the file- or directory belongs.

  This is not supported on Windows and defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

- `/+-GROUP:Alphabet:Pattern/  Sequence based on the group name`

  This returns a sequence ordered by the name of the group to which the file- or directory belongs.

  This is only supported on Windows if the `win32all` Python extensions are installed. Otherwise, this defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

- `/+-INODE:Alphabet:Pattern/  Sequence based on the inode number`

  This returns a sequence ordered by the file- or directory inode numbers.

  This is not supported on Windows and defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

- `/+-MODE:Alphabet:Pattern/  Sequence based on permissions`

  This returns a sequence ordered by the file- or directories permissions value.

- `/+-NLINK:Alphabet:Pattern/  Sequence based on the nlink count`

  This returns a sequence ordered by the number of links associated with the file- or directory.

  This is not supported on Windows and defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

- `/+-SIZE:Alphabet:Pattern/` `Sequence based on size`

  This returns a sequence ordered by the size of each file- or directory.

- `/+-UID :Alphabet:Pattern/` `Sequence based on the user ID number`

  This returns a sequence ordered by the ID number of the user that owns the file- or directory.

  This is not supported on Windows and defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

- `/+-USER:Alphabet:Pattern/` `Sequence based on user name`

  This returns a sequence ordered by the name of the user that owns the file- or directory.

  This is only supported on Windows if the `win32all` Python extensions are installed. Otherwise, this defaults to an alphabetic sequence equivalent to `/+-FNAME.../`.

COMMON TASKS AND IDIOMS

With a program as feature dense as **tren**, it's not possible to document every possible use case. The following examples cover many common applications of the program.

- Literal String Replacement

  Sometimes, all you want to do is replace a single substring in a name:

  ```
  tren.py -r Old=New OldHair-OldPeople # Yields: NewHair-OldPeople
  ```

- Replacing Or Removing All Instances Of A String

  Sometimes you want to replace every instance of the string:

  ```
  tren.py -i : -r Old=New OldHair-OldPeople # Yields: NewHair-NewPeople
  ```

- Changing A File's "Extension" Suffix

  Common where the suffix of a file is significant to an applications program:

  ```
  tren.py -i -1 -r .jpeg=.jpg *.jpeg
  ```

- Replace Spaces In A File Name With Underbars

  Even though spaces are allowed in file names in most OSs, they're a pain:

  ```
  tren.py -i : -r " "=_ *
  ```

- Appending- Or Prepending Strings To An Existing File Name

  Often, you want to keep the existing name, but add to it:

  ```
  tren.py -r =Prefix-/FNAME/
  tren.pu -r =/FNAME/-Suffix
  ```

- Ordering File Names By Last Modification Time

  It's nice to be able to see files in the order they were last modified. Usually, we preserve the old name when doing this:

  ```
  tren.py -r =/+MTIME::001/-/FNAME/
  ```

- Ordering File Names By Modification Time Within Date

  Sometimes, what we want is the order of modification *within* the date it was changed:

  ```
  tren.py -r =/MYEAR/-/MMON/-/MDAY/-/+MDATE::001/-/FNAME/ *
  ```

- Ordering File Names By Size

  This is handy if we want a directory listing to list the files in size order:

  ```
  tren.py -r =/SIZE/-/FNAME/ *
  ```

- Undoing A Previous Renaming

  In complex renamings, sometimes the only way to get back to your original names is to examine the renaming log. But in some cases it's pretty automatic:

  ```
  tren.py -r ='/$LOGNAME/'-/FNAME/ *
  ```

  This can be undone by:

  ```
  tren.py -r '/$LOGNAME/'-= *
  ```

  Generally, if you can isolate the text introduced by the previous renaming operation, and use it as the `old` string in another renaming request, this will work.

ODDS AND ENDS

- Quoting your command line arguments properly for the shell you use is critical. Things like spaces, \, and – have to be properly quoted or either the shell or **tren** itself are going to complain. Similarly, when using the `/$ENV/` and `/`cmd`/` renaming tokens, make sure to enclose them in single quotes if you're using a standard Unix shell like `bash`.
- Whitespace is almost always significant *within* a **tren** option. You'll need to put proper quoting around it to perserve if for **tren** to see, whether in a renaming request, an alphabet definition, or some part of a sequence renaming token.
- Quoting can also be tricky in include files. Remember that the contents of the include file are presented to **tren** as if they had been entered on the command line. For example, to replace spaces in a filename with underscores, we have to quote the space to preserve it as an argument to be passed to **tren**:

  ```
  # nospace: 'tren' include to get rid of spaces in filenames
  -i: -r' '=_  -r_-_=-  -i0
  ```

- Watch out for situations where an include file changes default or desired behavior. In the example above, the `-i:` is used to force replacement of *all* instances of spaces. The `-i0` at the end of the include resets **tren** to the default behavior of only replacing the first instance of a matching old string. That's fine if the include statement appears on the command line in a place where the default behavior was in force. But look what happens in a situation like this:

  ```
  tren.py -i3 -rx=y -Inospace -ra=b ....
  ```

  Prior to the include file being read, **tren** has been told to replace the 4th instance of a matching string. After the `nospace` include file has been read, this gets reset to replace the 1st instance of a matching old string. Make sure that's what you want for the `-ra=b` renaming request.

- Most shells don't care if you leave a space between an option and its argument. It's a really good idea to do so as a matter of habit, especially when dealing with a complex command line driven tool like **tren**.

- **tren** will attempt to do any requested renaming. However, if you manage to embed some character in the new name that the operating system doesn't like, the renaming will fail and you'll be notified of the fact. Notwithstanding the fact that you can do all manner of clever things with **tren**, some restraint is called for when constructing new file- or directory names.

- **tren** will prevent you from trying to rename something to a null string or a name too long for the operating system. Mostly this is not an issue *unless* you managed to concoct a renaming request that ends up requiring recursive backups. In that case, the backup suffix can be tacked onto the file name enough times that the file name becomes too long for the OS to catch. While **tren** can, and does catch this, **it cannot unwind what it has done thus far and you CAN LOSE FILES THIS WAY!!!**. The smart move here is to use test mode and make sure your proposed renaming isn't going to require deeply recursive backups.

- Save the output from your **tren** runs in logs. That way, if you have to unwind a renaming gone bad, you'll have a record of what was done.

- The use of `-bf` is **STRONGLY DISCOURAGED** and is provided only for the most sophisticated and careful users.

BUGS, MISFEATURES, OTHER

You must be running Python 2.6.x or later. **tren** makes use of features not supported in releases prior to this. **tren** has not been tested with Python 3.x and is presumed not to work with it until/unless otherwise demonstrated.

As a general matter, **tren** should run on any POSIX-compliant OS that has this version (or later) of Python on it. It will also run on many Microsoft Windows systems. If the Windows system has the `win32all` Python extensions installed, **tren** will take advantage of them for purposes of deriving the names of the user and group that own the file or directory being renamed.

This program is **EXPERIMENTAL** (see the license). This means its had some testing but is certainly not guaranteed to be perfect. As of this writing, it has been run on FreeBSD, Linux, Windows XP, and Mac OS X. It has not, however, been run on 64-bit versions of those OSs.

If you have experience, positive or negative, using **tren** on other OS/bitsize systems, please contact us at the email address below.

HOW COME THERE'S NO GUI?

tren is primarily intented for use by power users, sys admins, and advanced users that (mostly) find GUIs more of a nuisance than a help. There are times, however when it would be handy to be able to select the files to be renamed graphically. TundraWare has a freely available macro programmed file browser. It will work nicely in such applications:

http://www.tundraware.com/Software/twander/

COPYRIGHT AND LICENSING

For terms of use, see the `tren-license.txt` file in the program distribution. If you install **tren** on a FreeBSD system using the 'ports' mechanism, you will also find this file in `/usr/local/share/doc/tren`.

AUTHOR

Tim Daneliuk
tren@tundraware.com

DOCUMENT REVISION INFORMATION

`$Id: tren.rst,v 1.200 2010/11/17 19:52:56 tundra Exp $`

You can find the latest version of this program at:

http://www.tundraware.com/Software/tren

This document was produced using reStructuredText:

http://docutils.sourceforge.net/rst.html