PipeWrench User Guide

Table of Contents

Some Preliminaries

License Agreement

This program is free software. You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful; however, it and its related files are provided as is, without warranty of any kind, expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. In no event shall the author be held liable for any loss of profit or any other damages, including but not limited to direct, indirect, special, incidental, and consequential damages. Please see the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses.

Contacting the Author

For feedback, general inquiries and technical support questions regarding PipeWrench, please send an email to bwb@fireflysoftware.com. Please include “PipeWrench” in the subject line.

Introduction

PipeWrench is a powerful pipe-based text editing utility for Windows and Linux that allows you to transform text into other forms quickly.[1] It enables you to solve many “everyday” text-based problems without you having to address low-level issues like declaring variables or instantiating objects. All you do is simply combine filters together into little mini-programs called *pipes*. Each filter in a pipe applies some basic text processing function to the incoming text and then passes that processed text on to the next filter in the pipe for additional processing. A typical PipeWrench program or “pipe” consists of perhaps only a dozen such filters, (of course, your boss doesn’t have to know that).

PipeWrench’s 60+ text translation filters enable you to accomplish all sorts of text editing tasks—either interactively or from shell scripts. For example, you can search and replace text in web pages, convert between comma-delimited and fixed-width files, create mailing lists, extract email addresses, access log file data, format source code, convert text exported from one program for import into another program, automate interactive processes...

1 Under Windows, PipeWrench's GUI application is implemented using the “Winforms” API. Under Linux, it is implemented via GTK.

Features

- runs on both Windows and Linux platforms
- employs a rich set of 60+ text filters
- pipes can be constructed and debugged using PipeWrench's GUI application
- pipes can be run from the PipeWrench GUI, the terminal or from shell scripts
- pipes can be "called" with arguments passed to them
- a pipe's effects can be constrained to only a portion of the input text
- new filters can be added via *plugins*

Limitations

Like all software solutions, PipeWrench has its limits. The very design that makes PipeWrench a capable tool for cranking out solutions to many everyday text processing problems also limits its range of applications. For one thing, PipeWrench is only designed to process *text*. It isn't designed to handle *binary* content. PipeWrench expects its input to be formatted as lines of text; each line terminated by a "newline". In addition, PipeWrench is limited to working with just the amount of text that it can hold in memory at one time and it isn't intended for use in applications that require handling of more text than this. If you require such capability you may want to consider other available solutions.

How System Level Pipes and Filters Work

In Windows and Linux, any console program whose input can be redirected, using the input redirection operator, "<" to originate from a source other than the keyboard and whose output can be redirected, using the output redirection operator, ">" to go to a destination other than the *console* display is technically known as a *filter*. You are probably already familiar with the very useful filter called *sort*. It is often employed from the command prompt as follows:

```
sort <unsorted.txt
```

This sorts the text in *unsorted.txt* and outputs the resulting text to the console display. This is great if you only need one filter to process your text...

Following is a more capable form of the above example used on a Windows system (on Linux, substitute "cat" for the "type" command):

```
type unsorted.txt | sort
```

Here, the *piping* operator, "|" effectively connects the *output* of the *type* command with the *input* of the *sort* filter, creating what is known as a *pipe*. This example demonstrates a simple one-filter pipe that is functionally identical to the first example given above.

Pipes can be created containing any number of filters. For example, the *more* filter could be added to our pipe to pause each output screen of sorted text:

```
type unsorted.txt | sort | more
```

In the above examples output is being sent to the console display. A pipe's output can also be sent to a file using the output redirection operator, ">" as shown here:

```
sort <unsorted.txt >sorted.txt
```

Here, *sort* takes it's input from *unsorted.txt*, sorts it and writes it to the file, *sorted.txt*. If the file, *sorted.txt* already exists, it gets overwritten with the new data. If you merely want to append the new data to a file that already exists, use the append redirection operator ">>" instead as shown here:

```
sort <unsorted.txt >>sorted.txt
```

Here, the same is accomplished using the piping operator:

```
type unsorted.txt | sort >>sorted.txt
```

PipeWrench Pipes

The prior section describes generally how pipes and filters work at the command prompt on Windows and Linux systems (again, on Linux systems, replace the "type" command with "cat"). Text originates from some source, goes through one or more filters and is output either to the console display or redirected to a file.

PipeWrench takes this same paradigm of pipes and filters and simply wraps it in a graphical user interface in which the pipe is transposed from one that is laid out horizontally (as those depicted in the prior section) to one that is *vertical*. A PipeWrench pipe is thus presented to the user as multiple lines, each one representing a single filter in the pipe. A pipe that parses the words in the input text, pads each one to equal lengths and then joins them into 3 columns would therefore be formatted onto three lines as follows:

```
ParseWords
PadLinesRight ' '
JoinLines 3
```

This transposing of a pipe in the GUI version of PipeWrench however does not occur in PipeWrench's command line application, (PCL), which is used at the command line or from within shell scripts. With PCL, the above pipe would still be presented horizontally as shown by the following example:

```
type in.txt | pcl "ParseWords | PadLinesRight ' ' | JoinLines 3"
```

Seasoned "pipers" from either a Windows or Linux background may notice that the two piping operators ("|") following the invocation of PCL are contained within double quotes. Strange as it may seem, this is not a typo as PipeWrench pipes are processed *internally* by PipeWrench and it is only at their "ends" that system level piping takes place.

A Quick Tour of the GUI

A) The *Pipe* control is where the pipe is constructed or loaded from file for execution.

B) In addition to typing them, pipe commands can be entered into the *Pipe* control by clicking the commands found in the *Pipe Commands* list control.

C) The *Input Text* tab contains the *Input Text* control where text is entered or copied for processing.

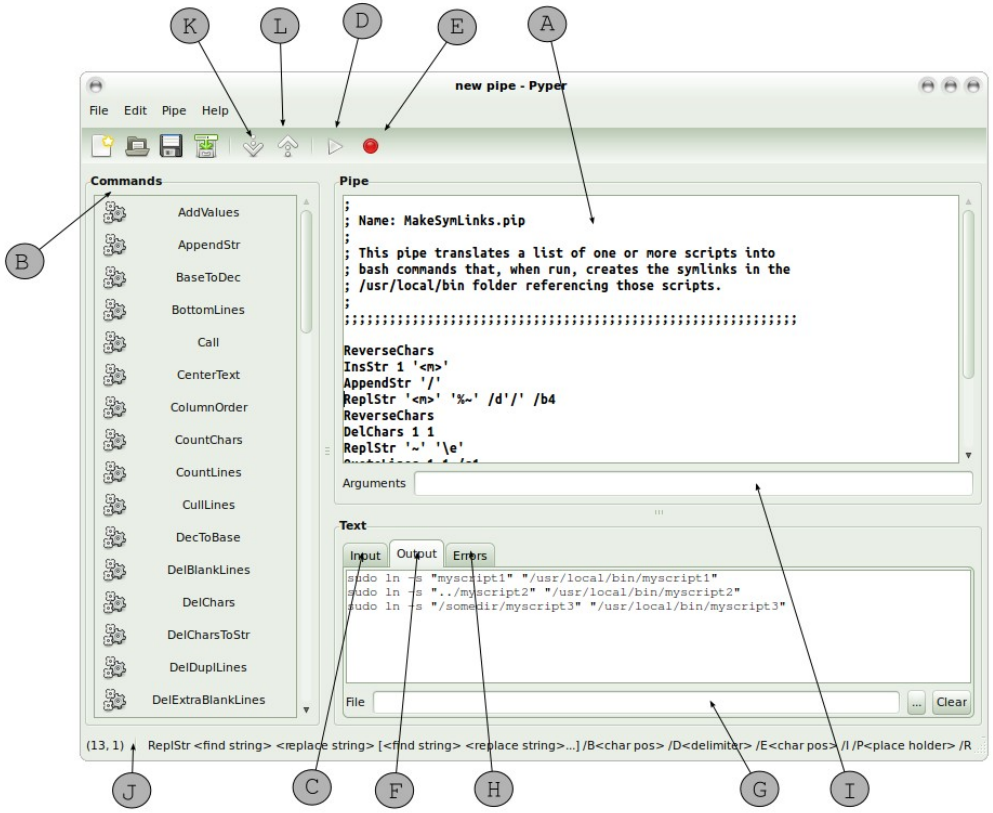
D) The *Run Pipe* button executes the pipe.

E) The *Run Pipe to Cursor* button executes the pipe only to the line containing the cursor.

F) The pipe's resulting text is output to the *Output Text* control found on the *Output Text* tab.

G) Alternately, the pipe's output text can be written directly to a file. Note that on the *Input Text* tab there exists a similar control that you can set to directly reference a file for reading input text and when you set it, this corresponding output control (G) is automatically set to the same file name with " (edited)" appended. This auto-generated name can be edited if you wish.

H) If there are any errors during execution, they'll be output to the *Errors* control on the *Errors* tab.



I) If a pipe requires arguments, they can be entered in the *Pipe Arguments* control. Note that arguments entered into the *Pipe Arguments* control must be formatted the same as if they were entered on the command line—that is, all string arguments must be single quoted.

J) The *statusbar* displays line / character position information as well as the command syntax for any filter located at the current line of the *Pipe* control.

K) A pipe can be imported into the *Pipe* control from the text clipboard using the *Import Pipe* button. This allows you to load a pipe into the PipeWrench GUI from a shell script (either for making changes to the pipe or for debugging).

L) The pipe currently loaded in the *Pipe* control can be exported to the text clipboard using the *Export Pipe* button. This allows a pipe that has just been developed (or debugged) in the PipeWrench GUI to be reformatted for use in a shell script.

Note: Any time that you need to enter a text column as an argument to a filter, you can determine the column value by placing the cursor into the *Input* or *Output Text* control (whichever applies) to see what column a given character is located at. As helpful as this is, there's a command on the *Edit* menu (*Insert Cursor Col*) to make this process easier still. When you're typing a filter command into a pipe that requires a text column be entered, if you'll leave the cursor where the inserted column value should go and then simply click in the applicable *Input/Output Text* control at the required column, you can then click the *Insert Cursor Col* menu command and the column's integer value will be inserted for you into the pipe's filter command at the last location where the cursor resided in the *Pipe* control.

Working With Pipes

Creating a New Pipe

To start a new pipe, clear any current *Pipe* control contents using the *New Pipe* button. If the current contents haven't been saved, you will be prompted to do so. Once the *Pipe* control is cleared, you can either add PipeWrench commands to the pipe by typing them directly into the *Pipe* control or, you can click on commands listed in the *Commands* control. Typing a space immediately following a command's name in the *Pipe* control will cause the command's syntax to be displayed in the *Status bar*.

Opening an existing Pipe

A pipe that has been previously saved to disk can be loaded into the *Pipe* control for execution (or for editing) by clicking the *Open* button.

Saving Pipe Changes

To save a pipe's changes to disk, click the *Save Pipe* button. If the pipe is newly created, you will be prompted for a file name. If the pipe is already saved and you just want to save it at a different location on disk, choose the *Save As* button instead to save the pipe. Again, you'll be prompted for the file name.

Importing a Pipe

A PipeWrench pipe (not to be confused with a system level pipe) can be "imported" from the command line or from a script by copying it (including the surrounding double quotes) and then by clicking the *Import Pipe* button. This will load the pipe into PipeWrench.

Exporting a Pipe

A pipe that is currently loaded into PipeWrench can be "exported" for use at the command line or in a script by clicking the *Export Pipe* button. The entire pipe will be copied and enclosed in double quotes, ready for pasting.

Executing a Pipe

If the *Pipe* control has a pipe loaded, that pipe can be executed by clicking the *Run Pipe* button.

Alternately, the pipe can be executed *partially*; that is, up to the line containing the cursor by clicking the *Run Pipe to Cursor* button. This feature allows you to examine the text output between the filters in a pipe.

Constructing Your First Pipe

The best way to learn how to use PipeWrench is to simply use it. To demonstrate how PipeWrench can be used to process text from one form to another, let's take the all-too-common situation where someone wants to rename a relatively large number of files on his computer without having to do a lot of typing. Suppose you downloaded the complete first season of the Andy Griffith Show onto your Linux computer (to help preserve the life of your beloved DVD collection, of course). Being as the files will ultimately be contained in a subfolder named "Andy Griffith" and being as you're not a big fan of long file names and redundancy, you decide the files should be named like "Manhunt.avi" rather than "Andy.Griffith-S01E02-Manhunt.avi".

To start, you generate a list of all of the files in the appropriate folder using the shell command, "ls -l":

```
-rw-r--r-- 1 johnd johnd 181271262 2013-07-30 16:15 Andy.Griffith-S01E01-New Housekeeper.avi
-rw-r--r-- 1 johnd johnd 181009578 2013-07-30 15:46 Andy.Griffith-S01E02-Manhunt.avi
-rw-r--r-- 1 johnd johnd 181325624 2013-07-30 16:38 Andy.Griffith-S01E03-Guitar Player.avi
...
```

Being the initial text which the pipe will operate on, you copy this text into the *Input Text* control of PipeWrench's GUI. Your goal of course is to transform this list of file names into a list of shell commands that, when executed, will result in the renaming of the files. Thus, as an example, you'd like the first line to be transformed into something functionally equivalent to the following (in Linux the "mv" command is used to rename a file or folder):

```
mv "Andy.Griffith-S01E01-New Housekeeper.avi" "New Housekeeper.avi"
```

Enclosing of the file names in double quotes is of course needed simply because some of the episode's file names may contain blanks. Now, take a moment to compare that first line's "before" and "after". Immediately we recognize that much of the text at the head of each of the input lines is irrelevant with regards to the output we wish to obtain. We're only interested in the text on each line that begins with "Andy". Everything prior to that can be deleted. However, rather than explicitly delete this unwanted text, we can simply re-order the text on each line so that this text is removed in the process of reordering. As seen in the "after" version of the first line, we need to obtain two copies of the file name and simply remove the extraneous text from the second copy. Again, this is accomplished by reordering the text on each line. Before we do that, we should first insert whatever additional text is required to produce the desired "mv" command. To that end, the first PipeWrench command we'll employ is the InsStr filter which we enter as the first line of the *Pipe* control in PipeWrench's GUI:

```
InsStr 53 'mv "'
```

This filter is used to simply insert the string “mv ”, followed by a double quote, at the beginning of the file name in each line of text (that is, at column 53). Clicking the *Run Pipe* button, results in the following:

```
-rw-r--r-- 1 johnd johnd 181271262 2013-07-30 16:15 mv "Andy.Griffith-S01E01-New Housekeeper.avi
-rw-r--r-- 1 johnd johnd 181009578 2013-07-30 15:46 mv "Andy.Griffith-S01E02-Manhunt.avi
-rw-r--r-- 1 johnd johnd 181325624 2013-07-30 16:38 mv "Andy.Griffith-S01E03-Guitar Player.avi
...
```

As you can see, the result is that the text, “mv ” followed by a double quote is inserted into each line just ahead of the text, “Andy”. Next, we want to append a double quote to the end of each line by adding an AppendStr filter to our pipe:

```
AppendStr '"'
```

Again, clicking the “run” button, the result of these two filters on the input text is now this:

```
-rw-r--r-- 1 johnd johnd 181271262 2013-07-30 16:15 mv "Andy.Griffith-S01E01-New Housekeeper.avi"
-rw-r--r-- 1 johnd johnd 181009578 2013-07-30 15:46 mv "Andy.Griffith-S01E02-Manhunt.avi"
-rw-r--r-- 1 johnd johnd 181325624 2013-07-30 16:38 mv "Andy.Griffith-S01E03-Guitar Player.avi"
...
```

Ignoring the extraneous text at the beginning of each line, we now have the beginning part of our “mv” command, which includes the file name we ’re renaming, enclosed in double quotes. Next, the reordering of each line is performed using the ReorderColumns filter:

```
ReorderColumns 53 102 56 102
```

This filter takes each line of text output by the filter before it (the AppendStr filter) and reorders the text so that only that text between columns 53 and 102 is included followed by the text between columns 56 and 102. As you might guess from the overlapping of the columns, this effectively duplicates the single file name found on each line of text output by the prior filter. The result is as follows:

```
mv "Andy.Griffith-S01E01-New Housekeeper.avi"     "Andy.Griffith-S01E01-New Housekeeper.avi"
mv "Andy.Griffith-S01E02-Manhunt.avi"             "Andy.Griffith-S01E02-Manhunt.avi"
mv "Andy.Griffith-S01E03-Guitar Player.avi"       "Andy.Griffith-S01E03-Guitar Player.avi"
...
```

At this point, if we tried to run these lines of text as shell commands in a Linux terminal, they would of course generate errors being as the “old” and “new” file names are identical. We still need to remove the unwanted text from the head of each of the “new” file names. This we can do using the DelChars filter:

```
DelChars 52 21
```

All this filter does is delete 21 characters beginning with character 52. Following is the final result:

```
mv "Andy.Griffith-S01E01-New Housekeeper.avi"     "New Housekeeper.avi"
mv "Andy.Griffith-S01E02-Manhunt.avi"             "Manhunt.avi"
mv "Andy.Griffith-S01E03-Guitar Player.avi"       "Guitar Player.avi"
...
```

This resulting list of shell commands can now be copied/pasted into the Linux terminal and executed in order to effect the renaming of the files. Whether we have 10 files to rename or 1000, it only takes a 4 filter pipe to create the Linux commands needed to accomplish this automated renaming:[2]

```
InsStr 53 'mv "'
AppendStr '"'
ReorderColumns 53 102 56 102
DelChars 52 21
```

2 Actually, if we employ a regular expression, then just one filter is required; but, as this is a demo intended to show how to construct pipes...

PipeWrench Command Line (PCL)

PipeWrench Command Line, (PCL) is the non-GUI version of PipeWrench meant to be executed at the command line or from shell scripts. PCL uses the same piping engine as its GUI counterpart and therefore a pipe created in the GUI version of PipeWrench can also be used with PCL.

The syntax for using PCL is as follows:

pcl <pipe specification> [<pipe arguments>]

A <pipe specification> is one or more filter specifications; each separated from the next by a pipe character, (|) and the entire collection enclosed in double-quotes. For example, the following is a pipe specification containing two filters:

```
"ParseWords | InsLineNo /w2 /z"
```

On a Linux system, this pipe specification could be used by PCL to *filter* the contents of the file, *in.txt*:

```
cat in.txt | pcl "ParseWords | InsLineNo /w2 /z"
```

Note that in Windows, you'd use the "type" command here instead of "cat". Note too that the piping character "|" inside of the PipeWrench pipe specification is *not* recognized as such by the shell's command interpreter as it is enclosed within the double-quotes and is therefore considered part of the string argument passed to PCL.

Following are some examples illustrating the use of PCL at the command-line or from a shell script:

```
echo 3,2,1,4 | pcl "ReplStr ',' '\e' | SortLines | AppendStr ',' | JoinLines | StripChars 1"

cat in.txt | sort | pcl "InsStr 1 ' ' | InsLineNo /w2 /z" >out.txt

pcl "SortLines" <in.txt >out.txt
```

Importing and Exporting Pipes

It's a simple matter in PipeWrench to convert back and forth between GUI and command line (PCL) pipes. Recall from earlier (in *Constructing Your First Pipe)* we created a pipe to transform a list of TV show episodes into shell commands that could be used to rename each episode:

```
InsStr 53 'mv "'
AppendStr '"'
ReorderColumns 53 102 56 102
DelChars 52 21
```

Had we wanted to employ this pipe from the command line or from inside of a shell script, we merely would have needed to "export" the pipe to the clipboard and then paste it to the appropriate destination. This is done by clicking the *Export* (up arrow) button on PipeWrench's toolbar and then by pasting (ctrl-v) the resulting text where the pipe is needed. If the above pipe was currently open in the PipeWrench GUI and we attempted this here, the following single line would be pasted (including the enclosing double quotes):

```
"InsStr 53 'mv #22' | AppendStr '#22' | ReorderColumns 53 102 56 102 | DelChars 52 21"
```

Here, I've added blanks before and after the "pipe" characters to improve readability. To use this pipe from inside of a shell script, you would only need to send it as a single argument to PCL. Assuming that the input text we wish to process is currently contained in the file *Input.txt*, you would do this:

```
cat Input.txt | pcl "InsStr 53 'mv #22' | AppendStr '#22' | ReorderColumns 53 102 56 102 | DelChars 52 21"
```

Notice that the double quotes originally referenced in our GUI pipe, have been converted to the ASCII sequence, "#22" which effectively "escapes" the quotes so that they cannot interfere with the shell's command line processing.

Importing a pipe from a script back into PipeWrench is done just as easily. If the above pipe specification enclosed in quotes were copied to the clipboard, you could simply click the *Import* (down arrow) button on PipeWrench's toolbar and the pipe would then be loaded into PipeWrench, ready for testing, debugging, etc. Note that some manual editing may be required if a script-based pipe contains extraneous text such as Bash variables.

Some Pitfalls

Following are some pitfalls to watch out for when using PipeWrench:

Direct editing of a text file using system level piping can't generally be done. That is, at the command line, (regardless of whether PipeWrench is involved or not), a file cannot be piped back into itself like this:

```
type a_file.txt | sort >a_file.txt
```

Doing so causes the contents of the file (here, a_file.txt) *to be cleared*. To get around this problem, you can change the above construct so that a temporary file is used:

```
type a_file.txt | sort >temp.txt
type temp.txt > a_file.txt
```

Input / output controls are limited in the amount of text that they can handle. If you are working with a rather large amount of text in the PipeWrench GUI application, you may find that the text controls are incapable of handling it (the exact limit depending on your platform of choice). The solution is to make use of PipeWrench's ability to access text files directly (see item "G" in *A Quick Tour of the GUI*). As you might expect, this problem can also be resolved by making use of PipeWrench's command line application, PCL.

String Parameters

The string parameters used by various PipeWrench filters must be single-quoted strings. To include a single quote into a single-quoted string, you must place two of them together, thus the string, `''''` (four adjacent single quotes) represents a *single* quote mark. For example, the string,

```
Isn't life grand?
```

would be represented as:

```
'Isn''t life grand?'
```

At times it may be necessary to specify other characters in a string which cannot by referenced directly. PipeWrench allows you to do this by entering the character, "#" into the string followed by a two digit, (hexadecimal) value which all together, represents an ASCII character. To include the character, "#" into a string, two of them must appear together. The string, "`#0D#0A`" represents a carriage return linefeed pair. Note that the hexadecimal value following each "#" character in a string must contain exactly 2 digits. *The leading zero cannot be omitted*. The above string could therefore be represented as:

```
'Isn#27t life grand?'
```

It's also possible to represent ASCII characters using the backslash form. For example, the above could also be written as:

```
'Isn\x27t life grand?'
```

Additionally, some special characters may be represented directly using a backslash (and two consecutive backslashes would be used to represent a single one):

newline (\n)
platform independent end-of-line (\e)
tab character (\t)
backslash character (\\)

See also *Appendix D: The ASCII Character Set*

Regular Expressions

Regular expressions can be supplied as string parameters for a number of PipeWrench filters. In general, any string parameter of a filter that can be used to “match” text can be treated as a *regular expression* pattern. For a list of elements that can be used to build regular expression patterns, see *Appendix C: Regular Expressions Reference*.

To illustrate the use of regular expressions in PipeWrench, consider the problem of having to rename a list of files so that they are effectively archived in-place. In particular, suppose that we wanted to rename the following list of files so that each resulting filename ends with “_old.txt”:

```
a_file.txt
another_file.txt
yet_another_file.txt
```

One solution would be to do the following:

```
QuoteLines 1 1 /s1
ReorderColumns 1 40 1 40
JustCharsRight
InsStr 1 'mv' 76 '_old'
DelExtraBlanks<bwb>
```

The result of running this pipe is a list of shell command that could be run from the terminal to rename the files:

```
mv "a_file.txt" "a_file_old.txt"
mv "another_file.txt" "another_file_old.txt"
mv "yet_another_file.txt" "yet_another_file_old.txt"
```

Though this wasn’t that difficult to accomplish, a much more “elegant” solution would be to use a regular expression. Such a solution requires but a single filter:

```
ReplStr '^(.+)(\\.txt)' 'mv "$0" "$1_old$2"' /r
```

Running this pipe produces the exact same results as before.

Note that two backslashes (\\) are required in the regex pattern because both PipeWrench and the regex engine interpret them.

Sets

In PipeWrench, a *set* is a collection of lines in the input text that are related to one-another according to their line positions. For example, the following text consists of 3 groups of 4 sets of data:

```
             --> 54667
            |    Tiny Tim   <--
            |    05/05/1980    |
            |    M             |
 Set #1     --> 87896         |
(Zipcode)   |    John Doe   <--  Set #2
            |    01/15/1960    |  (Name)
            |    M             |
            --> 98798          |
                Lex Luther <--
                05/26/1940
                M
```

In this example, the 4 sets of data are *zipcode*, *name*, *birthdate* and *sex*. If the number of sets is 4, as shown here, then a set consists of every 4 lines—lines 1,5,9... all belong to the first set and lines 2,6,10... all belong to the second, etc.

Various PipeWrench filters take advantage of sets thus making it possible to work with groupings of text such as tabular data (where each table column would represent an individual set).

Interface Templates

PipeWrench communicates with both filters and called pipes via their command interfaces. A command interface consists of the parameters and switches that are passed to a command. Parameters can be either whole numbers or single-quoted strings. Switches can communicate additional numeric or string data in addition to simple, on/off or “boolean” states.

In order to syntax-check a command interface at runtime, PipeWrench uses an *interface template*. This template may be composed of the following symbols:

Symbol	Meaning
n	a numeric parameter
s	a quoted string parameter
/?n	a switch with a numeric value (where “?” represents any single alpha character “A”-“Z”)
/?s	a switch with a quoted string value (where “?” represents any single alpha character “A”-“Z”)
/?	a logical or boolean switch (where “?” represents any single alpha character “A”-“Z”)

To help convey how interface templates are constructed, following are some standard PipeWrench filters and the templates used to implement them:

Filter	Template	Description
AppendStr	s /P	Appends (or prepends) the given string to each input line.
InsStr	n s [n s...]	Inserts one or more specified strings at a specified character position.
JoinLines	[n] /P	Joins every given number of input lines onto an output line, optionally preserving paragraphs. If the number of lines parameter is omitted, all input lines are joined onto one output line.
SplitLines	n [n...]	Splits each input line at one or more character positions.
TopLines	n	Outputs up to the specified number of lines from the “top” of the text.

Note: All template symbols—whether regular parameter or switch—must be separated from one another by a space, “[”, “]” or “.”. Those parameters enclosed inside of “[” and “]” are optional and any such optional part of a template must follow all required parameters (those not enclosed in “[” and “]”). The three dots (...) are used to specify that an optional parameter or a group of optional parameters (everything enclosed in “[” and “]”) can repeat. Switches must follow the required and optional parameters and they must be preceded by a blank.

For called pipes, an interface template must be provided via the first “comment block” located at the top of the pipe:

```
; Name:     RangeOfLines
; Template: n n
; Prompt:   <first line> <last line>
; Desc:     Returns a range of lines
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

The only requirement is that this block of comments (the pipe *header*) contains a “Template:” tag with corresponding template; however, the pipe developer is encouraged to include the additional tags shown as they provide good minimal documentation and a future release of PipeWrench may make use of them.

Note: This comment block or header must begin with the *first* line of the pipe. If you receive the error, “Parameter or switch is extraneous” when calling a pipe, check to be sure there are no blank lines prior to this block.

Calling Pipes

Any pipe that you create in PipeWrench and save to disk can be called by another pipe. For such a pipe to receive arguments passed to it, it must include an *interface template* that documents the pipe's command interface (see *Interface Templates*). In addition, the pipe's body must contain argument placeholders that correspond to the calling parameters that are specified in this interface. Called pipes cannot have switches passed to them; however, numeric or string arguments may be passed that can be used to construct switches for passing to the pipe's filters. For an example illustrating the calling of pipes, see the Call command.

Argument placeholders are used to bind the arguments that are passed to a called pipe. They are of the form "[" + <arg no> + "]" where <arg no> is the ordinal number of the corresponding argument in the argument list. Such placeholders are replaced by their corresponding arguments (*not* including surrounding quotes if the argument is a string). As those representing strings do not include surrounding quotes, they can be embedded within other PipeWrench strings and thus easily combined with other text. For instance, given the string, "A" is passed as the first argument to a pipe, the following filter in the pipe would replace any occurrences of "*" found with "LBL-A":

```
ReplStr '*' 'LBL-[1]'
```

Note: Placeholders that have no corresponding argument are left as-is. This allows you to reference "[1]", for example, in a PipeWrench string *literally* so long as arguments are not passed to the pipe. In situations where you must literally reference a bracketed number that is also a valid placeholder for a pipe, you can reference it using an ASCII sequence. For example, "[1]" could be referenced as "#5b1]". Likewise, it could be referenced as "\[1]". It's good practice to do this when referencing bracketed numbers that aren't intended as placeholders just in case arguments are later added to the pipe.

To illustrate the calling of a pipe, consider the following simplistic example. Given a pipe that has been saved to disk:

```
; Name: test.pip
; Template: n s
 InsStr [1] '[2]'
```

If you were to call this pipe from the PipeWrench GUI, you'd have to specify arguments for the two parameters in the *Pipe Arguments* control. But you could just as easily call it from a pipe utilized at the command line like this:

```
echo -e "now\nis\nthe\ntime" | pcl "call 'test.pip' 1 'word: '"
```

The resulting output would be:

```
word: now
word: is
word: the
word: time
```

Debugging Pipes

Debugging pipes in PipeWrench is a matter of examining the text results between filters. In this way you can easily see where any problem is originating from. To debug a pipe, enclose the filters whose output you're interested in with the two commands, SetDebugOn and SetDebugOff. When the pipe is run, any text output from a filter will be sent to a debug file (whose location on disk can be changed in *Edit/Preferences*). For example, given the following input text,

```
the quick brown fox
```

the pipe,

```
SetDebugOn
ParseWords
AppendStr '*'
JoinLines
SetDebugOff
```

would generate the following debug output:

```
ParseWords

the
quick
brown
fox

AppendStr '*'

the*
quick*
brown*
fox*

JoinLines

the*quick*brown*fox*
```

In this example, the ParseWords filter first parses each word onto separate lines. The AppendStr filter appends an asterisk (*) to the end of each line and the JoinLines filter then joins all lines onto one line.

Note: The intermediate text results between filters can also be viewed using the *Run Pipe to Cursor* button.

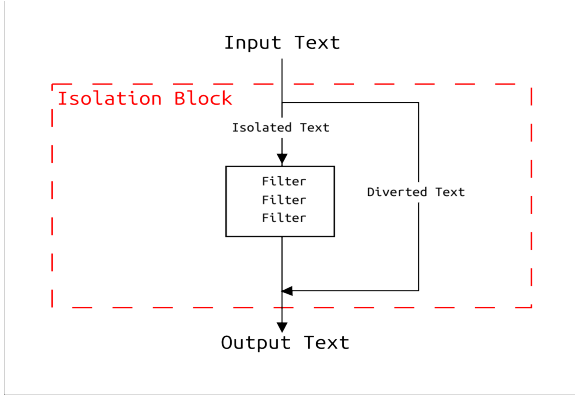
Isolation Blocks Overview

Isolation blocks constrain a group of pipe commands so that they operate only on a targeted portion of the input text rather than the entire input text which would otherwise be the case.

An isolation block consists of three parts:

- the IsolateLines command
- one or more pipe commands (filters, called pipes, etc.)
- the EndIsolate command

The IsolateLines command outputs only the text that is targeted according to the command's parameters (see the IsolateLines command for details of how this works). This "isolated" text is thus made available for processing by all subsequent filters in the pipe—up to the corresponding EndIsolate command. Once the matching EndIsolate command is encountered, the text—which has been processed by each of the filters inside the isolation block—is then re-combined with the text that originally surrounded it. This re-combined text—some of which was edited and some which was not—is then output to the rest of the pipe by the EndIsolate command. In effect, the targeted text is passed *through* the filters in the isolation block whereas the text not targeted by the isolation block is diverted *around* them:



Isolation blocks allow you to perform multiple distinct edits of the input text from within a single pipe. Because they isolate the text being operated on, a single pipe can actually edit different parts of the input text *differently*. As a simple example of how isolation blocks are used, consider the problem of simply joining the middle three lines of the following text onto one line and then uppercasing the characters on that line:

```
Isolation blocks allow a pipe to operate
on an isolated
section of text
and perform
multiple distinct edits of the overall text.
```

Without isolation, this would be tedious at best. With the help of isolation on the other hand, it is a trivial matter. The following pipe does just what we want:

```
IsolateLines 'isolated' /e'perform'

   ; Append a blank to each line:

   AppendStr ' '

   ; Join the lines and uppercase them:

   JoinLines
   UpperCase
EndIsolate
```

It first isolates the targeted text defined by the two boundary strings, “isolated” and “perform” which consists of the three short lines. Once isolated, this text is then joined onto a single line and uppercased, giving us the desired output:

```
Isolation blocks allow a pipe to operate
ON AN ISOLATED SECTION OF TEXT AND PERFORM
multiple distinct edits of the overall text.
```

The power of isolation blocks is that they allow you to edit the isolated text *without affecting other text*. Isolation blocks can even be *nested*. As you might imagine, this is desirable in situations where you need to edit selected lines in an already isolated section of text.

Note: For additional examples illustrating isolation blocks, see IsolateLines.

Appendix A: PipeWrench Command Reference

The following are PipeWrench's standard commands:

AddValues	adds two or more numbers found in each input line
AppendStr	concatenates the supplied string to the end of each line of the input text
BaseToDec	converts numbers of the given base found in the input text to decimal
BottomLines	outputs the given number of lines from the end of the text
Call	calls a pipe from the currently executing pipe
CenterText	centers the input text in a field of the given character width
ColumnOrder	a pre-filter to the JoinLines filter which allows columns to be ordered down instead of across
CountChars	outputs the number of characters
CountLines	outputs the number of lines
CullLines	removes groups of lines encountered in the input text
DecToBase	converts decimal numbers found in the input text to another base
DelBlankLines	removes blank lines from output
DelChars	deletes characters from each line at specified character positions
DelCharsToStr	deletes characters until string is encountered
DelDuplLines	acts on sorted lists removing all duplicate lines
DelExtraBlankLines	removes extraneous blank lines
DelExtraBlanks	removes extraneous blanks from each line
DivValues	performs division with two numbers in the input text
EndIsolate	used with IsolateLines to constrain pipe commands to an isolated block of text

ExclLines	excludes all lines from output that contain the specified string
ExtractLines	extracts groups of lines encountered in the input text
FoldLines	acts on sorted lists folding duplicate lines
GroupLines	groups lines together that contain a string
InclLines	includes all lines in the output that contain the specified string
InsLineNo	inserts a line number at the specified character position of each line
InsStr	inserts character strings into each line at specified character positions
IsolateLines	used along with EndIsolate to constrain pipe commands to isolated block of text
JoinLines	joins every n lines of text into a single line of text
JustCharsLeft	left justifies characters
JustCharsRight	right justifies characters
LeftChars	returns the given number of characters from the beginning of each line of text
LinesByPos	outputs lines according to their position
LowerCase	converts uppercase characters to lowercase
MultValues	multiplies two or more numbers in the input text
OutDuplLines	outputs lines that are duplicated in the text
OverlayChars	overlays each line with character strings at specified character positions
PadLinesLeft	pads each line on the left to the given character width with the given character string
PadLinesRight	pads each line on the right to the given character width with the given character string
ParseCSV	parses quoted, comma-delimited fields onto separate lines
ParseWords	parses the text into individual words
QuoteLines	surrounds lines with quotes
ReorderColumns	re-arranges the column order of each line

ReplStr	replaces character strings found in the text
ReverseChars	reverses the characters in each line of the input text
RightChars	returns the given number of characters from the end of each line of text
RotCharsLeft	rotates characters left given no of places
RotCharsRight	rotates characters right given no of places
RotCharsToStr	rotates each line until given string is at its beginning
SetDebugOn	configures pipe debugging on
SetDebugOff	configures pipe debugging off
ShiftChars	shifts text into specified character positions
SortLines	sorts the text
SpliceFile	combines text from a text file to the input text
SplitLines	splits each line at the given character position(s)
StripChars	deletes a given number of characters from the end of each line
SubValues	subtracts two numbers found on each line of text
TopLines	outputs the given number of lines from the beginning of the input text
TotalColumns	totals columns of numeric values
TrimLines	removes leading and trailing white space from each line of text
TrimLinesLeft	removes leading white space from each line of text
TrimLinesRight	removes trailing white space from each line of text
UpperCase	converts lowercase characters to uppercase
WrapText	wraps text at given character position

AddValues

syntax: AddValues [<char pos>...] /I<ins pos> /W<width> /D<decimals> /S
summary: adds two or more numbers found in each input line

The AddValues filter adds two or more numbers found in each input line and outputs the result of the addition per each line. Each parameter supplied is the character position of a number to be added and in each case it must point at the left-most digit of the number to be added or at the white space prior to it. Alternately, if no parameters are specified then each input line of text is expected to contain numeric values separated by whitespace. If the /I switch is specified, the result of each addition is inserted back into the line of text at the specified character position rather than simply being output alone, one per line. The /W switch determines the width of the resulting numeric values and the /D switch determines the number of decimal places displayed after the decimal point. If /S is specified, the resulting values will be output in scientific notation.

Given input text containing the four columns of numbers...

```
2.5       3.25      6.25      6.5
----+----1----+----2----+----3----+----4----+----5
```

the pipe...

```
AddValues 1 11 21 31
```

outputs the result of each addition:

```
18.50
----+----1----+----2----+----3----+----4----+----5
```

Adding the /I switch...

```
AddValues 1 11 21 31 /i41
```

causes the result of the addition to be inserted back into each line, in this case at character position 41:

```
2.5       3.25      6.25      6.5       18.50
----+----1----+----2----+----3----+----4----+----5
```

See also *SubValues*, *MultValues*, *DivValues*

AppendStr

syntax: AppendStr <string> /P
summary: concatenates the supplied string to the end of each line of the input text

The AppendStr filter concatenates the supplied string to the end of each line of the input text and then outputs the resulting lines. If the /P switch is specified then the string is instead pre-pended to each line of the input text.

Given the following two lines of text...

```
one
two
```

the pipe...

```
AppendStr ' line'
```

will output:

```
one line
two line
```

See also *InsStr*

BaseToDec

syntax: BaseToDec <radix> /I<ins char pos> /S<scan char pos> /W<width> /Z
summary: converts numbers of the given base found in the input text to decimal

The BaseToDec filter converts whole numbers of the given base, (2 - 62) found in the input text to decimal, (base 10) and then outputs the resulting values. The only required parameter is the radix of the numeric values found in the input text. If an insert character position is specified using the /I switch the resulting decimal value will be inserted into the output text at the specified character position rather than being output alone. The /S switch can be used to specify the character position where scanning for the input values begins. If the /S switch is omitted, scanning begins at character position 1. The /W switch determines the width of the numeric result. The default is 6. The /Z switch causes the value(s) to be displayed with leading zeros.

The characters read from standard input must follow the numerical sequence (dependent on radix):

“0” - “9”, “A” - “Z”, “a” - “z”

Because this sequence contains both upper and lower case characters, case is obviously significant when considering numbers to be converted. As such, even if they were of the same base, the values “f2ac” and “F2AC” would not be equivalent!

To illustrate the BaseToDec filter, the binary value...

```
10011
```

run through the pipe...

```
BaseToDec 2
```

will output:

```
    19
----+----1----+----2
```

The BaseToDec filter can be used in conjunction with the DecToBase filter to perform base conversions from one base to another assuming that they are both in the range, (2 - 62). See the DecToBase filter for an example of how this is done.

See also *DecToBase*

BottomLines

syntax: BottomLines <no of lines>
summary: outputs the given number of lines from the end of the text

The BottomLines filter outputs the given number of lines from the bottom of the input text. It can be used in conjunction with the TopLines filter to return a range of lines from the middle of the input text. For example, given the following input text...

```
this is line 1
this is line 2
this is line 3
this is line 4
```

the pipe...

```
TopLines 3
BottomLines 2
```

will output:

```
this is line 2
this is line 3
```

For another way of returning a range of lines, see the LinesByPos filter.

See also *TopLines*, *LinesByPos*

Call

syntax: Call <pipe file name> [<arg>...]
summary: calls a pipe from the currently executing pipe

The Call command is used to invoke another pipe from the currently executing one. It takes as its first argument a single-quoted string representing the called pipe's filename:

```
Call 'c:\pipes\mypipe.pip' ...
```

Just like with filters, pipes can accept arguments (though not switches). In a Call command, any such arguments that follow the called pipe's filename are passed to the called pipe. The arguments that are expected by a called pipe is determined by its template. This template is recorded in the comment block at the top of the pipe (its "header"). To illustrate the calling of a pipe, we start with a pipe that's been saved to disk as "Envelop.pip":

```
; Name: Envelop.pip
; Template: s s
;
; This pipe encloses each line
; inside the given two strings.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

InsStr 1 '[1]'
AppendStr '[2]'
```

As you can see, the pipe's template is "s s". This tells PipeWrench that the pipe requires two string arguments. It could therefore be called from another pipe as shown here:

```
; Name: MainPipe.pip
;
; This pipe parses the given text onto
; separate lines, envelops each line
; inside two strings and then uppercases
; the text.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

ParseWords
Call 'Envelop.pip' '<<<' '>>>'
UpperCase
```

If this pipe were run against the text...

```
one two three
```

following would be the results:

```
<<<ONE>>>
<<<TWO>>>
<<<THREE>>>
```

Note: If the called pipe is referenced without a path, (i.e. “mypipe.pip”) it must exist in the same folder as the calling pipe.

See also *Calling Pipes*

CenterText

syntax: CenterText <field width>
summary: centers the input text in a field of the given character width

The CenterText filter centers the input text in a field of the given character width by padding on the left with the appropriate number of blank characters.

Given the input text...

```
This
Is
Centered
```

the pipe...

```
CenterText 50
```

results in:

```
                      This
                        Is
                    Centered
----+----1----+----2----+----3----+----4----+----5
```

See also *JustCharsLeft*, *JustCharsRight*

ColumnOrder

syntax: ColumnOrder <no of rows> <no of columns>
summary: a pre-filter to the JoinLines filter which allows columns to be ordered down instead of across

The ColumnOrder filter is used as a pre-filter to the JoinLines filter so that the text stream can be reformed into columns that are ordered top to bottom instead of being ordered left to right as would normally be output from the JoinLines filter, given a sorted list of items. The first parameter required is the number of rows that you wish to limit the table to. The second is the number of columns.

To illustrate, suppose you wanted to format a list of unordered names into 3 columns, ordered. Given the list of unordered names...

```
pat
jim
cindy
ernie
stephanie
brian
bert
susan
randy
leonard
philip
terry
kathy
evan
jacob
```

the pipe...

```
SortLines
PadLinesRight ' ' /w15
JoinLines 3
```

will output three columns of names:

```
bert           brian          cindy
ernie          evan           jacob
jim            kathy          leonard
pat            philip         randy
stephanie      susan          terry
----+----1----+----2----+----3----+----4----+----5
```

Notice that the names are ordered across the columns, from left to right. Had we intended for them to be ordered down each column instead, we would need to apply the ColumnOrder filter ahead of the JoinLines filter:

```
SortLines
PadLinesRight ' ' /w15
ColumnOrder 5 3
JoinLines 3
```

This produces three columns of names that are ordered down each column:

```
bert           jacob          philip
brian          jim            randy
cindy          kathy          stephanie
ernie          leonard        susan
evan           pat            terry
----+----1----+----2----+----3----+----4----+----5
```

Note: The number of lines being joined by the JoinLines filter must be the same as the number of columns specified in the ColumnOrder filter.

As a second example of using the ColumnOrder filter, suppose you had a list of names and addresses that you needed to create mailing labels from:

```
Jim Philips              1290 Pin Oak Ln          Westhaven, FL 77392
Ted Crowder              7382 West 21st St.       Lincoln, AR 72882
Lisa Taylor              1213 Pompano St.         Norden, TX 77283
Laura Michelson          1550 Fenn Way Blvd.      Treeville, CO 77662
Peter Piper              8399 Fourth St.          Ashton, CT 72882
William Orson            7113 France Ave.         Linden KY 66383
Susan Simpson            728 Conner Peak St.      Freemont IN, 45892
----+----1----+----2----+----3----+----4----+----5----+----6----+----7
```

If your label stock is one label wide, you could simply execute the following pipe:

```
AppendStr '\e'
SplitLines 26 51
```

Running it would result in the desired output:

```
Jim Philips
1290 Pin Oak Ln
Westhaven, FL 77392

Ted Crowder
7382 West 21st St.
Lincoln, AR 72882

Lisa Taylor
1213 Pompano St.
Norden, TX 77283

Laura Michelson
1550 Fenn Way Blvd.
Treeville, CO 77662

Peter Piper
8399 Fourth St.
Ashton, CT 72882
...
```

The AppendStr filter is used to first double-space the list of names and the SplitLines filter then splits each line into three separate lines. Note that the SplitLines filter doesn't affect the blank lines as their length, (0) is less than the column positions 26 and 51 at which splitting takes place.

Now, suppose you wanted the labels to be printed on a laser printer, 3 across by say, 15 deep per page. Adding three more filters to the original pipe gives us the pipe:

```
AppendStr '\e'
SplitLines 26 51
PadLinesRight ' ' /w25
ColumnOrder 12 3
JoinLines 3
```

Its execution displays the following output:

```
Jim Philips              Laura Michelson          Susan Simpson
1290 Pin Oak Ln          1550 Fenn Way Blvd.      728 Conner Peak St.
Westhaven, FL 77392      Treeville, CO 77662      Freemont IN, 45892

Ted Crowder              Peter Piper
7382 West 21st St.       8399 Fourth St.
Lincoln, AR 72882        Ashton, CT 72882

Lisa Taylor              William Orson
1213 Pompano St.         7113 France Ave.
Norden, TX 77283         Linden KY 66383
----+----1----+----2----+----3----+----4----+----5----+----6----+----7
```

In this particular example, I limited the output to 3 labels deep by specifying 12 rows in the ColumnOrder filter instead of the 28 that would be required for a single column, (7 labels x 4 lines per label). This allowed the output to span 3 columns for the sake of demonstration.

CountChars

syntax: CountChars /I<char pos> /L /W<width> /Z
summary: outputs the number of characters

The CountChars filter outputs the total number of characters in the input text. If the /L switch is specified then the total number of characters per each line is output instead. If the /I switch is specified along with the /L switch then each line's total is inserted back into the line at the specified character position. The /W switch determines the width of the resulting numeric value(s) and the /Z switch causes them to be displayed with leading zeros. Given the following text...

```
The CountChars filter
outputs the total number
of characters in
the input text.
```

the pipe...

```
CountChars
```

will return:

```
    76
----+----1
```

Note: If the /I switch is specified without the /L switch, it will be ignored.

See also *CountLines*

CountLines

syntax: CountLines /W<width> /Z
summary: outputs the number of lines

The CountLines filter outputs the total number of lines in the input text. The /W switch determines the width of the resulting numeric value and the /Z switch causes the count to be displayed with leading zeros. Given the following text...

```
The CountLines filter
outputs the total number
of lines in
the input text.
```

the pipe...

```
CountLines /w3 /z
```

will return:

```
004
----+----1
```

See also *CountChars*

CullLines

syntax: CullLines <begin string> <end string> /A /I /R
summary: removes groups of lines encountered in the input text

The CullLines filter removes from output the first group of lines encountered in the input text in which the initial line of the group contains the <begin string> and the last line of the group contains the <end string>. If the /A switch is specified, all such groups encountered in the input text will be removed. If the /I switch is specified, case is ignored in locating the "boundary" strings in the input text. If the /R switch is specified the strings are treated as regular expression patterns. If both strings are found on the same line, then only that line will be removed. Given the following text...

```
the
quick brown
fox
jumped
over the lazy
dog and
the lazy
dog
didn't even bark
```

the pipe...

```
CullLines 'fox' 'lazy'
```

will result in:

```
the
quick brown
dog and
the lazy
dog
didn't even bark
```

Note: The CullLines filter performs the opposite function of the ExtractLines filter which outputs the text between the two strings instead of removing the text.

See also *ExtractLines*

DecToBase

syntax: DecToBase <radix> /I<ins char pos> /S<scan char pos> /W<width> /Z
summary: converts decimal numbers found in the input text to another base

The DecToBase filter converts decimal, (base 10) numbers found in the input text to another base, (2 - 62) and then outputs the resulting values. The only required parameter is the radix of the numbers being converted to. If an insert character position is specified using the /I switch the resulting decimal values will be inserted into the output text at the specified character position rather than being output alone. The /S switch can be used to specify the character position where scanning for the input (decimal) values begins. If the /S switch is omitted, scanning begins at character position 1. The /W switch determines the width of the numeric result. The /Z switch causes the value(s) to be displayed with leading zeros.

Given a single input line of text containing the characters “255” (interpreted as a decimal value), the following pipe will convert it to “FF”, (base 16 or hexadecimal):

```
DecToBase 16
```

As the filters, BaseToDec and DecToBase can convert to and from decimal, they can be combined to allow conversion from any base to any other base, (2 - 62). For example, the following pipe could be used to convert the value “1010” (in binary) to “A” (hexadecimal):

```
BaseToDec
DecToBase 16
```

See also *BaseToDec*

DelBlankLines

syntax: DelBlankLines
summary: removes blank lines from output

The DelBlankLines filter filters all blank lines. A blank line is considered to be one that contains no characters. A line containing just blank or tab characters is NOT considered a blank line. In order to remove lines containing only blank or tab characters, the TrimLines or TrimLinesRight filters must first be used to remove the whitespace.

Given the following 3 lines of text -- the 2nd of which is blank...

```
This line is not blank

This line is also not blank
```

the pipe...

```
DelBlankLines
```

will output the non-blank lines only:

```
This line is not blank
This line is also not blank
```

See also *DelExtraBlankLines*

DelChars

syntax: DelChars <char pos> <no of chars> [<char pos> <no of chars>...]
summary: deletes characters from each line at specified character positions

The DelChars filter deletes a specified number of characters from each line at a given character position. Multiple deletions may be performed, however the position parameters must be in ascending order.

To illustrate the DelChars filter, suppose you wanted to delete the 2nd and 4th columns from the following fixed-width data:

```
54667  Tiny Tim     05/05/1980  M   20456.45   2347281749
87896  John Doe     01/15/1960  M   234888.56  6482610982
98798  Lex Luther   05/26/1940  M   9834.01    4566281737
67491  Betty Boop   05/26/1932  F   34772.01   1902630722
----+----1----+----2----+----3----+----4----+----5----+----6
```

The pipe...

```
DelChars 8 13 33 4
```

delivers the desired results:

```
54667  05/05/1980  20456.45   2347281749
87896  01/15/1960  234888.56  6482610982
98798  05/26/1940  9834.01    4566281737
67491  05/26/1932  34772.01   1902630722
----+----1----+----2----+----3----+----4----+----5----+----6
```

Note: To reorder the columns, see the ReorderColumns filter.

See also *StripChars*, *ReorderColumns*

DelCharsToStr

syntax: DelCharsToStr <string> /I /N<count> /R
summary: deletes characters until string is encountered

The DelCharsToStr filter removes characters from a line until the given string, if found, is relocated to the beginning of the line. If the string is not found in a line, no editing is done to that line. DelCharsToStr can accept a /N switch specifying the number of strings to "index". By default, this is 1. If you specify a count that is greater than the number of occurences of that string in a line, then the line will not be edited. If the /I switch is specified the case of the strings is ignored. If the /R switch is specified the strings are treated as a regular expression patterns.

Like with ShiftChars and RotCharsToStr, the DelCharsToStr filter can be very useful for editing all the lines of input text as a whole (rather than each line individually). This is done by preprocessing the text onto a single line first using the JoinLines filter. Suppose you wanted to remove the third line of the following text:

```
This is line one.
This is line two.
This is line three.
This is line four.
```

This can be achieved using the DelCharsToStr filter using the following pipe:

```
AppendStr '<eol>'
JoinLines
InsStr 1 '<bot>'
RotCharsToStr '<eol>' /n2
DelCharsToStr '<eol>' /n2
RotCharsToStr '<bot>'
ReplStr '<bot>' '' '<eol>' '\e'
```

Granted, this could be done more easily using the ExclLines filter; however, there are times when the above technique is better suited; for example, when manipulating free-form text where "lines" of text are not so well defined, (i.e. HTML). For an example, see the demo, "Edit HTML Files".

See also *ShiftChars*

DelDuplLines

syntax: DelDuplLines [<char pos> <char pos>] /A /D<delimiter> /F /I
summary: acts on sorted lists removing all duplicate lines

The DelDuplLines filter acts on sorted lists, removing any extraneous duplicate lines (thus, by default, outputting only a single line for each set of duplicate lines). If the /A switch is specified, all lines are removed from each set of duplicate lines (thus only unique lines are output). If the /F switch is specified, only the first line of each set of duplicates is removed. If the /I switch is specified, case is ignored in determining uniqueness. If a delimiter is specified using the /D switch, then duplication is based on the text at the beginning of each line up to the delimiter. You can also specify a range of character positions upon which duplication is determined.

See the OutDuplLines filter for an example using the DelDuplLines filter.

Note: In order for the DelDuplLines filter to work, the input text must be sorted. If removing duplicates based on a range of character positions, the input text must be sorted on that same range of character positions.

Note: Either a range or a delimiter should be specified, but not both.

See also *SortLines*, *OutDuplLines*

DelExtraBlankLines

syntax: DelExtraBlankLines
summary: removes extraneous blank lines

The DelExtraBlankLines filter removes extraneous blank lines. It replaces any number of consecutive blank lines with a single one. Note that to be considered blank, a line must contain absolutely no characters, not even blanks or tab characters; therefore, it may be necessary to remove trailing blanks and tabs from all lines first, (using either the TrimLines or TrimLinesRight filters) before the DelExtraBlankLines filter is used.

See also *DelBlankLines*

DelExtraBlanks

syntax: DelExtraBlanks
summary: removes extraneous blanks from each line

The DelExtraBlanks filter removes all extraneous blanks from each line. It replaces any number of consecutive blanks with a single one.

Note: The DelExtraBlanks filter has no effect on tab characters.

DivValues

syntax: DivValues <char pos> <char pos> /I<ins char pos> /W<width> /D<decimals> /S
summary: performs division with two numbers in the input text

The DivValues filter divides two numbers found in each input line and outputs the result of the division per each line. The two required parameters specify at what character positions the numbers involved in the division are located and in each case must point at the leftmost digit of the number or at the white space prior to it. If the /I switch is specified, the result of the division is inserted back into the line of text at the specified character position rather than simply being output alone, one per line. The /W switch determines the width of the resulting numeric values and the /D switch determines the number of decimal places displayed after the decimal point. If /S is specified, the resulting values will be output in scientific notation.

Given input text containing the two columns of numbers:

```
100  20
----+----1----+----2
```

the pipe...

```
DivValues 1 5
```

will result in:

```
  5.00
----+----1----+----2
```

Adding the /I switch...

```
DivValues 1 5 /i10
```

causes the result of the division to be inserted back into each line, in this case at character position 10:

```
100  20    5.00
----+----1----+----2
```

See also *AddValues*, *SubValues, MultValues*

EndIsolate

syntax: EndIsolate
summary: used with IsolateLines to constrain pipe commands to an isolated block of text

The EndIsolate filter is used along with IsolateLines to constrain pipe commands to an isolated block of text. For an example, see IsolateLines.

See also *IsolateLines*

ExclLines

syntax: ExclLines <string> [<begin char pos> <end char pos>] /I /R
summary: excludes all lines from output that contain the specified string

The Excllines filter filters all lines that contain the specified string. If the optional range of character positions is specified, then lines are excluded only if the string exists within that range of character positions. If the /I switch is specified then the case of the string is ignored. If the /R switch is specified the string is treated as a regular expression pattern.

For example...

```
ExclLines 'the' 5 7
```

will remove all lines from output that contain the string “the” in character positions 5 through 7.

See also *InclLines*

ExtractLines

syntax: ExtractLines <begin string> <end string> /A /I /R
summary: extracts groups of lines encountered in the input text

The ExtractLines filter extracts the first group of lines encountered in the input text in which the initial line of the group contains the <begin string> and the last line of the group contains the <end string>. If the /A switch is specified, all such groups encountered in the input text will be extracted. If the /I switch is specified, case is ignored in locating the "boundary" strings in the input text. If the /R switch is specified the strings are treated as regular expression patterns. If both strings are found on the same line, then only that line will be extracted. Given the following text...

```
the
quick brown
fox
jumped
over the lazy
dog and
the lazy
dog
didn't even bark
```

the pipe...

```
ExtractLines 'fox' 'lazy'
```

will result in:

```
fox
jumped
over the lazy
```

Note: The ExtractLines filter performs the opposite function of CullLines which removes the text between the two strings instead of passing the text.

See also *CullLines*

FoldLines

syntax: FoldLines [<char pos> <char pos>] /D<delimiter> /E /I /J<join opt> /W<width> /Z
summary: acts on sorted lists folding duplicate lines

The FoldLines filter acts on sorted lists, folding any duplicate lines (outputting just a single line for each group of duplicate lines along with an appended count). If a range of character positions is specified then uniqueness is determined based on that range only. Likewise, if a delimiter character is specified using the /D switch, uniqueness is determined based on the text found at the beginning of each line up to the delimiter. If the /I switch is specified, case is ignored in determining uniqueness. If the /E switch is specified, input lines are expected to already have a count appended (by default, input lines have an implicit count of "1"). A join option can be specified using the /J switch. A join option of 0 (the default) causes output of a single line plus a count for each group of duplicate lines. Use of join option 1 causes each group of duplicate lines to be joined onto one line. Join option 2 simply combines join options 0 and 1.

Given some sorted text as follows...

```
one
one
two
three
three
three
four
four
```

the pipe...

```
PadLinesRight ' '
AppendStr ' '
FoldLines
```

returns:

```
one         2
two         1
three       3
four        2
```

See also *JoinLines*

GroupLines

syntax: GroupLines <string> [<begin char pos> <end char pos>] /I /R
summary: groups lines together that contain a string

The GroupLines filter groups all lines containing the given string at the end of the text. It is primarily used as a “pre-filter” to the IsolateLines filter which expects all lines being isolated (based on their containing a particular string) to be contiguous. If the optional character position range is specified, then only that segment of each input line of text is considered when searching for the string. If the /I switch is specified then the case of the string is ignored. If the /R switch is specified the string is treated as a regular expression pattern.

Note: The functionality provided by the GroupLines filter can be provided by the SortLines filter if the string that determines whether or not a line is to be isolated exists at the same character position for all lines you wish to target for isolation.

For examples that illustrate the GroupLines filter, see IsolateLines.

InclLines

syntax: InclLines <string> [<begin char pos> <end char pos>] /I /R
summary: includes all lines in the output that contain the specified string

The InclLines filter outputs all lines that contain the specified string. If the optional range of character positions is specified, then lines are output only if the string exists within that range of character positions. If the /I switch is specified then the case of the string is ignored. If the /R switch is specified the string is treated as a regular expression pattern.

For example, the pipe...

```
InclLines 'the' 5 7
```

will output all lines that contain the string “the” in character positions 5 through 7.

See also *ExclLines*

InsLineNo

syntax: InsLineNo /L<init no> /I<incr> /P<ins pos> /S<no of lines> /W<width> /Z
summary: inserts a line number at the specified character position of each line

The InsLineNo filter inserts a line number into each line. The line number is inserted at the beginning of each line unless the /P switch specifies that the line number is to be inserted at another character position. You can use the /L switch to specify the initial line number which can be a negative number. By default it is 1. Specify a /I switch if you want the line numbers to increment by some integer value other than the default which is 1, (it can be a negative number as well). Specify a /S switch if you want the line number sequence to repeat every specified number of lines (usually to represent members of a *set*). The default is 0 meaning that no repetition occurs. The /W switch determines the width of each line number. If not specified, the default numeric width, 6 is used. If the /Z switch is specified, the line numbers will be left-padded with zeros.

As an example, supposed we had the following list of names...

```
John
Susan
Paul
George
```

and for whatever reason, we wanted to vertically “flip” them. This is easily done using the InsLineNo filter in combination with SortLines:

```
InsLineNo /w6 /z
SortLines /r
DelChars 1 6
```

Execution of this pipe results in the expected output:

```
George
Paul
Susan
John
```

Note: The InsLineNo filter is typically used as a pre-filter to record the current “order” of the input text prior to subsequent processing by other filters which may reorder the text as part of their processing. This allows the lines of text to later be sorted back to their original order. For an example where the InsLineNo filter is used for this purpose see the IsolateLines filter.

InsStr

syntax: InsStr <char pos> <string> [<char pos> <string>...]
summary: inserts character strings into each line at specified character positions

The InsStr filter inserts a character string into each line at the specified character position. Multiple insertions into each line may be performed however the character position parameters must be in ascending order.

Given the following list of batch files...

```
convert.bat
this.bat
that.bat
theother.bat
```

the pipe...

```
InsStr 1 'c:\\temp\\' 15 '"a command line argument"'
```

will output:

```
c:\temp\convert.bat   "a command line argument"
c:\temp\this.bat      "a command line argument"
c:\temp\that.bat      "a command line argument"
c:\temp\theother.bat  "a command line argument"
----+----1----+----2----+----3----+----4----+----5
```

See also *AppendStr*, *OverlayChars*

IsolateLines

syntax: IsolateLines <string> [<begin char pos> <end char pos>] /Es /I /R
summary: used along with EndIsolate to constrain pipe commands to isolated block of text

The IsolateLines command is used along with the EndIsolate command to limit the effects of an enclosed block of pipe commands to just an isolated portion of the input text. Such a block of pipe commands is known as an *isolation block* and it can consist of filters, called pipes and even nested isolation blocks. Isolation blocks enable a PipeWrench pipe to perform *multiple, distinct edits to its input text*—something that isn't possible when using O/S managed pipes.

The text to be isolated by IsolateLines can be established by one of two methods:

- A group of lines can be isolated based on whether or not the lines in it contain a given string (optionally located between two character positions). For this method to work, the targeted lines of text must be grouped together and they must comprise the first such group found in the input text. Any lines that are not part of this initial group will not be targeted for isolation. Typically, this doesn't present a problem given that one usually wants to isolate *all* lines in the input text that contain the given string and not just a subset of them. By using the GroupLines filter prior to the IsolateLines filter, one can guarantee that all lines containing the necessary string will be grouped together.

- A block of text can be isolated based on its first line containing a "beginning" <string> and its last line containing an "ending" string (as specified using the /E switch). Again, both the "beginning" and "ending" strings may have to be located between the two optionally provided character positions. Also, note that both strings may be found on the same line. Unlike with the first method, isolating a block of text this way does not require use of the GroupLines filter. So long as the beginning and ending strings select the intended block of text, IsolateLines can isolate it.

If the /I switch is specified then the case of the specified string(s) is ignored in establishing the isolated text. If the /R switch is specified, either string is treated as a regular expression pattern.

As a first example of isolating text, let's consider a simple, though not necessarily obvious case. IsolateLines can be used to isolate just a *single* line of text given that the supplied string uniquely selects that one line. In such case, there's no need to group lines prior to use of the IsolateLines filter since the line is already uniquely selected. For example, given the following three lines of text and our desire to insert missing line "three"...

```
one
two
four
```

the pipe...

```
IsolateLines 'two'
   AppendStr '\ethree'
EndIsolate
```

gives us the following:

```
one
two
three
four
```

Now, for a more significant example, suppose we have three "records" of data, each one consisting of 4 fields: *name*, *age*, *city* and *state*:

```
John L. Doe    42 Miami       FL
Frank N. Stein 20 Los Angeles CA
Norma Lee      16 Austin      TX
----+----1----+----2----+----3----+----4
```

Now, suppose we wanted to uppercase each person's name, *leaving all of the other fields unchanged*. This is a simple matter although the targeted text (the names) must be isolated first. Text isolation requires that *lines* being targeted be distinguishable from other *lines*. The key word here is "lines" and thus, our first order of business is to parse the fields onto separate lines using the SplitLines filter. Afterwards, we'll add two line numbers—the first one repeating every four lines (1,2,3,4,1,2,3,4...) and the second one simply ranging from 1-n (where "n" is the number of text lines).

The pipe thus far is as follows:

```
SplitLines 16 19 31
InsLineNo /w1 /s4
InsLineNo /w2 /z
```

When run, it outputs the following:

```
011John L. Doe
02242
033Miami
044FL
051Frank N. Stein
06220
073Los Angeles
084CA
091Norma Lee
10216
113Austin
124TX
```

Note that the two line numbers are shown here in *reverse* of the order that they were added—the first one added being the single digit line number in column 3 and the second one added taking up columns 1 and 2. The purpose of the single digit line number in column 3 is to make those lines containing a person's name distinguishable from all other lines. As you can see, every line containing a name can now be distinguished by the digit "1" found in column 3. The purpose of the second line number that occupies columns 1 and 2 is to record the order of the lines at this point in our pipe because otherwise, we'd lose this ordering as a result of the GroupLines filter that the pipe is next going to employ.

Following is the pipe with this filter added:

```
SplitLines 16 19 31
InsLineNo /w1 /s4
InsLineNo /w2 /z
GroupLines '1' 3 3
```

When it is run we get this output:

```
02242
033Miami
044FL
06220
073Los Angeles
084CA
10216
113Austin
124TX
011John L. Doe
051Frank N. Stein
091Norma Lee
```

Notice that the lines containing a person's name are now grouped at the end of the text. This makes it possible for the lines to be isolated. Next up comes the actual isolating of the lines. This only requires adding three more filters to the pipe:

```
SplitLines 16 19 31
InsLineNo /w1 /s4
InsLineNo /w2 /z
GroupLines '1' 3 3
IsolateLines '1' 3 3
   UpperCase
EndIsolate
```

Now, when run, we see that only those lines with the names are uppercased:

```
02242
033Miami
044FL
06220
073Los Angeles
084CA
10216
113Austin
124TX
011JOHN L. DOE
051FRANK N. STEIN
091NORMA LEE
```

After the text has been isolated and edited, all that remains is to "reverse" the changes to the text that were initially made in order to prepare for isolation. That means sorting the lines of text back to their original order, removing the added line numbers and then translating the lines of text back to their original columnar form. Following is the completed (even commented) pipe with these additions made to it:

```
; Place each "datum" onto a separate line:

SplitLines 16 19 31

; Uppercase only the text in column one:

InsLineNo /w1 /s4
InsLineNo /w2 /z
GroupLines '1' 3 3
IsolateLines '1' 3 3
   UpperCase
EndIsolate
SortLines
DelChars 1 3

; Translate the rows of text back to columns:
```

```
TrimLines
PadLinesRight ' ' /s4
AppendStr ' '
JoinLines 4
```

When run, the pipe generates the output desired:

```
JOHN L. DOE    42 Miami       FL
FRANK N. STEIN 20 Los Angeles CA
NORMA LEE      16 Austin      TX
----+----1----+----2----+----3----+----4
```

JoinLines

syntax: JoinLines [<no of lines>] /P
summary: joins every n lines of text into a single line of text

The JoinLines filter combines all lines in the input text onto a single line, in-effect removing all end-of-lines from the input text. If the optional number of lines parameter is specified then that many lines will be combined onto each line output. The /P switch is used to join all of the lines within each "paragraph" (a sequence of non-blank lines) onto a single line and is ignored if the <number of lines> parameter is specified.

Warning: JoinLines is the key to some of the more advanced piping techniques; however, be forewarned that it is costly from a performance standpoint. Try to limit the number of lines of text that are input to the JoinLines filter whenever possible by filtering unneeded text lines prior to its execution.

To illustrate use of JoinLines, consider the following text which represents three data records, each containing a name, age, city and state:

```
JOHN L. DOE
42
Miami
FL
FRANK N. STEIN
20
Los Angeles
CA
NORMA LEE
16
Austin
TX
```

Normally, you wouldn't encounter data records in this form; however, it is quite common to do so when working with pipes. To transform the above text to the more conventional form, you'd employ the following pipe:

```
AppendStr ','
JoinLines 4
StripChars 1
```

When executed, it results in the following comma-delimited data:

```
John L. Doe,42,Miami,FL
Frank N. Stein,20,Los Angeles,CA
Norma Lee,16,Austin,TX
```

Note: For a more practical example in which fixed-width data is translated to comma-delimited data (which uses the JoinLines filter), see the QuoteLines filter.

As another example illustrating the JoinLines filter, say you have multiple paragraphs of text:

```
This is the
first paragraph of text
contained on three lines.

This is the second
paragraph of text contained on two lines.

This is the third paragraph of text contained on one line.
```

Now, suppose you wanted to remove all end-of-lines just from each paragraph, without joining together *all* lines of text.

The following pipe does the trick:

```
AppendStr ' '
JoinLines /p
```

When run, it outputs:

```
This is the first paragraph of text contained on three lines.

This is the second paragraph of text contained on two lines.

This is the third paragraph of text contained on one line.
```

Note: For an example in which the JoinLines filter is used to create multiple columns of text from a single column, see the ColumnOrder filter.

JustCharsLeft

syntax: JustCharsLeft [<begin char pos> <end char pos>]
summary: left justifies characters

The JustCharsLeft filter left-justifies each line of text. If the optional range of character positions is specified, the filter operates only on that range of characters. For an example of how this works, see JustCharsRight.

See also *JustCharsRight*.

JustCharsRight

syntax: JustCharsRight [<begin char pos> <end char pos>]
summary: right justifies characters

The JustCharsRight filter right-justifies each line of text. If the optional range of character positions is specified, the filter operates only on that range of characters. The JustCharsRight filter can be used in combination with the PadLinesRight filter to right-justify entire lines:

```
PadLinesRight ' '
JustCharsRight
```

Optionally, JustCharsRight can justify a range of character positions. For example, if you wanted to right-justify just the folder names in the following text...

```
06/02/2002  02:39p  <dir>  Documents and Settings  Archived
10/28/2002  10:16a  <dir>  Outlook Mail Messages   Archived
05/12/2003  05:52p  <dir>  Program Files           Archived
04/14/2003  08:36a  <dir>  Trash                   Archived
04/01/2003  01:55p  <dir>  WINNT                   Archived
----+----1----+----2----+----3----+----4----+----5----+----6
```

you could call JustCharsRight like this:

```
JustCharsRight 28 49
```

This would result in the following:

```
06/02/2002  02:39p  <dir>  Documents and Settings  Archived
10/28/2002  10:16a  <dir>   Outlook Mail Messages  Archived
05/12/2003  05:52p  <dir>           Program Files  Archived
04/14/2003  08:36a  <dir>                  Trash  Archived
04/01/2003  01:55p  <dir>                  WINNT  Archived
----+----1----+----2----+----3----+----4----+----5----+----6
```

See also *JustCharsLeft*.

LeftChars

syntax: LeftChars <no of chars>
summary: returns the given number of characters from the beginning of each line of text

The LeftChars filter outputs the specified left-most characters from each input line of text.

Given the text...

```
123.45, Table
061.22, Chair
----+----1----+----2
```

the pipe...

```
LeftChars 6
```

gives:

```
123.45
061.22
----+----1----+----2
```

See also *RightChars*

LinesByPos

syntax: LinesByPos <begin line> <end line> [<begin line> <end line>...] /S<no of sets>
summary: outputs lines according to their position

The LinesByPos filter outputs those lines that fall within the specified range of lines. Multiple ranges can be specified but they must be in ascending order. For example, 1 3 7 10 16 16 would result in output of lines 1-3, 7-10 and 16. As seen from this example, single lines can be output only by specifying them twice; that is, as a range of one line. If the /S (set) switch is specified then the input source is not viewed as a single continuum of lines 1-n, but rather, as multiple, repeating groups of lines; i.e., 1-<no of sets>, 1-<no of sets>, 1-<no of sets> ... In such case, all specified ranges of lines are output from each group. To illustrate the LinesByPos filter, suppose you wanted to exclude unwanted fields from the comma-delimited text:

```
54667,Tiny Tim,05/05/1980,M
87896,John Doe,01/15/1960,M
98798,Lex Luther,05/26/1940,M
```

You might want, for example, to exclude the first and last fields of each line leaving only the person's name and birth date. To do this, first you need to parse each line's fields onto separate lines. One way to accomplish this is by using the ReplStr filter to replace the commas with end-of-lines:

```
ReplStr ',' '\e'
```

Execution of this one-filter pipe gives us the following:

```
54667
Tiny Tim
05/05/1980
M
87896
John Doe
01/15/1960
M
98798
Lex Luther
05/26/1940
M
```

Note: The ReplStr filter will not work if the comma-delimited data includes quoted strings that can contain commas. In such a case the ParseCSV filter is recommended instead.

Next, in order to exclude all fields except name and birth date, the LinesByPos filter is added:

```
ReplStr ',' '\e'
LinesByPos 2 3 /s4
```

This gives us:

```
Tiny Tim
05/05/1980
John Doe
01/15/1960
Lex Luther
05/26/1940
```

Finally, we add three more filters:

```
AppendStr ','
JoinLines 2
StripChars 1
```

These added filters restore the commas, join the lines (in groups of 2) and then strip off the extraneous comma on the end of each line, giving us the output we desire:

```
Tiny Tim,05/05/1980
John Doe,01/15/1960
Lex Luther,05/26/1940
```

Outputting a range of lines from a list according to position can also be accomplished by use of the TopLines and BottomLines filters. See BottomLines for an example of this.

See also *BottomLines*, *TopLines*

LowerCase

syntax: LowerCase
summary: converts uppercase characters to lowercase

The LowerCase filter converts uppercase characters to lowercase.

See also *UpperCase*

MultValues

syntax: MultValues [<char pos>...] /I<ins char pos> /W<width> /D<decimals> /S
summary: multiplies two or more numbers in the input text

The MultValues filter multiplies two or more numbers found in each input line and outputs the result of the multiplication per each line. Each parameter supplied is the character position of a number to be multiplied and in each case it must point at the left-most digit of the number to be multiplied or at the white space prior to it. *Note: if you want all numbers found on each line to participate in the multiplication, you need not specify any parameters at all*. If the /I switch is specified, the result of each multiplication is inserted back into the line of text at the specified character position rather than simply being output alone, one per line. The /W switch determines the width of the resulting numeric values and the /D switch determines the number of decimal places displayed after the decimal point. If /S is specified, the resulting values will be output in scientific notation.

Given the four columns of numbers...

```
3      2      4     10
2      6      3.2   1.8
----+----1----+----2----+----3----+----4
```

the pipe...

```
MultValues 1 5 15 20
```

will result in:

```
240.00
69.12
----+----1----+----2----+----3----+----4
```

Adding the /I switch...

```
MultValues 1 5 15 20 /I30
```

causes the result of the multiplication to be inserted back into each line, in this case at character position 30:

```
3      2      4     10       240.00
2      6      3.2   1.8       69.12
----+----1----+----2----+----3----+----4
```

Note: Again, in each example above, we could have omitted the character position arguments (1, 5, 15 and 20) and the result would have been identical.

See also *AddValues*, *SubValues*, *DivValues*

OutDuplLines

syntax: OutDuplLines [<begin char pos> <end char pos>] /D<delimiter> /I
summary: outputs lines that are duplicated in the text

The OutDuplLines filter acts on sorted lists and outputs only those lines that are duplicated in the input text. If the optional range of character positions is specified then uniqueness is determined based on that range only. If a delimiter is specified using the /D switch, then uniqueness is based on the text at the beginning of each line up to the delimiter. If the /I switch is specified, case is ignored in determining uniqueness.

Note: In order for the OutDuplLines filter to work, the input text must be sorted. If outputting duplicate lines based on a range of character positions, the input text must be sorted on that same range of character positions.

Note: Either a range or a delimiter should be specified, but not both.

To illustrate, given the following text...

```
1   40   now
2   34   is
3   26   the
4   40   time
5   48   for
6   32   all
7   26   good
8   88   men
9   40   to
----+----1----+----2
```

suppose you wanted to obtain a listing of all lines that were duplicated based on the 2-digit numbers in the second column. The pipe,

```
SortLines /p5
OutDuplLines 5 6
```

… would result in the desired output:

```
7   26   good
3   26   the
1   40   now
4   40   time
9   40   to
----+----1----+----2
```

If, however you just wanted to know *which* 2-digit numbers were duplicated, you would add an extra couple of filters to the original pipe, resulting in this:

```
SortLines /p5
OutDuplLines 5 6
DelDuplLines 5 6
ReorderColumns 5 6
```

Executing it would result in the unique list as follows:

```
26
40
```

See also *DelDuplLines*, *SortLines*

OverlayChars

syntax: OverlayChars <char pos> <string> [<char pos> <string>...]
summary: overlays each line with character strings at specified character positions

The OverlayChars filter overlays or “patches” each line at the specified character position with a string. Multiple overlays can be performed at the same time.

Given the text...

```
23.6 123 19.77
12.1 223 20.50
----+----1----+----2
```

the pipe...

```
OverlayChars 5 ',' 9 ','
```

gives:

```
23.6,123,19.77
12.1,223,20.50
----+----1----+----2
```

See also *InsStr*, *AppendStr*

PadLinesLeft

syntax: PadLinesLeft <pad string> /W<pad width> /S<no of sets>
summary: pads each line on the left to the given character width with the given character string

The PadLinesLeft filter pads each line on the left with the given character string to the width of the longest line. If the /W switch is specified then all lines are padded to that width instead. If the /S switch is specified then each line is padded to the width of the longest line in its set.

Given the text...

```
the
quick brown
fox
jumped
over the lazy
dog and
the lazy
dog
didn't even bark very much
```

the pipe...

```
PadLinesLeft '*'
```

outputs each line padded on the left with asterisks to the width of the longest line:

```
**********************the
***************quick brown
**********************fox
*******************jumped
*************over the lazy
*******************dog and
******************the lazy
**********************dog
didn't even bark very much
```

Adding the /S switch gives us the pipe:

```
PadLinesLeft '*' /s3
```

Its execution outputs each individual line padded on the left with asterisks to the width of the longest line in each of the *three* sets:

```
*****the
**quick brown
**********************fox
**jumped
over the lazy
******************dog and
the lazy
**********dog
didn't even bark very much
```

Why would you want to do this you ask? One reason for wanting to pad lines in this way is in preparation for outputting them as columns. Granted, that could be done without using the /S switch but using it allows the columns to be output will a minimum of space between them.

To output the text as three columns then, we might want to employ this pipe:

```
PadLinesLeft ' ' /s3
AppendStr '|'
JoinLines 3
StripChars 1
```

It's execution gives us three closely-spaced columns of text, each one right-justified:

```
     the|  quick brown|                     fox
  jumped|over the lazy|                  dog and
the lazy|          dog|didn't even bark very much
----+----1----+----2----+----3----+----4----+----5
```

For clarity sake, a pipe character (|) is used here to separate each column. Note that each resulting column is only as wide as is necessary. If you were to run this pipe without the /s3 switch on the PadLinesLeft filter, you'd see that both columns one and two are the same width as column three.

Note: The number of sets referenced in the PadLinesLeft filter and the number of lines referenced in the JoinLines filter must be the same for the columns to output properly. To increase the number of output columns simply increase the number of sets in the PadLinesLeft filter and the number of lines in the JoinLines filter.

Note: The /S switch is ignored if a /W switch is specified.

See also *PadLinesRight*

PadLinesRight

syntax: PadLinesRight <pad string> /W<pad width> /S<no of sets>
summary: pads each line on the right to the given character width with the given character string

The PadLinesRight filter pads each line on the right with the given character string to the width of the longest line. If the /W switch is specified then all lines are padded to that width instead. If the /S switch is specified then each line is padded to the width of the longest line in its set.

Given the text...

```
the
quick brown
fox
jumped
over the lazy
dog and
the lazy
dog
didn't even bark
```

the pipe...

```
PadLinesRight '*'
```

outputs each line padded on the right with asterisks to the width of the longest line:

```
the*************
quick brown*****
fox*************
jumped**********
over the lazy***
dog and*********
the lazy********
dog*************
didn't even bark
```

For an example illustrating use of the /S switch, see the PadLinesLeft filter.

Note: The /S switch is ignored if a /W switch is specified.

See also *PadLinesLeft*

ParseCSV

syntax: ParseCSV /Q<quote> /D<delimiter>
summary: parses quoted, comma-delimited fields onto separate lines

The ParseCSV filter parses comma-separated data values onto separate lines. Any surrounding quotes are left intact and by default, they are expected to be double-quotes; however, the character expected for quotes can be changed using the /Q switch. Likewise, by default the delimiter character expected to separate fields is a comma but it can be changed using the /D switch. If a quoted string contains embedded quotes then they must be in pairs:

```
"""Tiny"" Tim Thompson"
```

Note: ParseCSV expects that data values are either quoted or unquoted. Unquoted values are generally ordinal / numeric values like 56, 1.23 or true. Because whitespace is allowed both before and after delimiters, unquoted data values cannot contain embedded whitespace (actually, they can contain whitespace but it will be removed).

Suppose you wanted to translate the following quoted, comma-delimited data to fixed-width:

```
"54667","""Tiny"" Tim","05/05/1980","M",20456.45,"2347281749"
"98798","Luther, Lex","05/26/1940","M",9834.01,"4566281737"
"67491","Betty Boop","05/26/1932","F",34772.01,"1902630722"
----+----1----+----2----+----3----+----4----+----5----+----6----+----7
```

The first step in doing this is to parse all of the comma-separated fields onto separate lines. You could use the ReplStr filter to replace all commas (,) with an end-of-line, in-effect forcing all fields onto separate lines; however, any of the string fields could contain a comma (as for example, the name “Luther, Lex” in the second line) and therefore this isn’t an ideal solution. A better approach is to use the ParseCSV filter instead.

Sending the above data through the one-filter pipe...

```
ParseCSV
```

outputs each field onto a separate line:

```
"54667"
"""Tiny"" Tim"
"05/05/1980"
"M"
20456.45
"2347281749"
"98798"
"Luther, Lex"
"05/26/1940"
"M"
9834.01
"4566281737"
"67491"
"Betty Boop"
"05/26/1932"
"F"
34772.01
"1902630722"
```

Adding 3 more filters to the pipe gives us this:

```
ParseCSV
QuoteLines 1 6 /s6 /u
PadLinesRight ' ' /s6
JoinLines 6
```

The /U switch used by the QuoteLines filter is needed to unquote each line, (i.e. remove the outer quotes surrounding each line and then restore any embedded quotes). Finally, the parsed and unquoted fields are padded with blanks using PadLinesRight and then re-joined into fixed-width lines using JoinLines. The final output is:

```
54667"Tiny" Tim 05/05/1980M20456.452347281749
98798Luther, Lex05/26/1940M9834.01 4566281737
67491Betty Boop 05/26/1932F34772.011902630722
----+----1----+----2----+----3----+----4----+----5
```

For an example in which fixed-width data is translated to quoted, comma-delimited, see the QuoteLines filter. For an example that illustrates excluding fields from comma-delimited data (in which ParseCSV can be used) see the LinesByPos filter.

ParseWords

syntax: ParseWords /D<delimiter>
summary: parses the text into individual words

The ParseWords filter parses each “word” of the input text onto a separate line. By default, a “word” is any text that is delimited by blanks. If you want other characters to be considered as delimiters, you must provide a delimiter string using the /D switch.

Given the text...

```
Hurry!... Time's a wastin!
```

the pipe...

```
ParseWords
```

will produce the following output:

```
Hurry!...
Time's
a
wastin!
```

With additional delimiters specified via the /D switch...

```
ParseWords /D' !.'''
```

the output becomes:

```
Hurry
Time
s
a
wastin
```

QuoteLines

syntax: QuoteLines <begin line> <end line> [<begin line> <end line>...] /B /D<delimiter> /O<option> /Q<quote> /S<no of sets> /U
summary: surrounds lines with quotes

The QuoteLines filter is a special purpose filter used to translate fixed-width data to quoted, comma-delimited data. It surrounds those lines that fall within the specified range with quotes and replaces each embedded quote with a pair of quotes. If the /B switch is specified then embedded quotes are replaced by a backslash character (\) followed by a quote instead of two quotes. By default, quotes are expected to be double-quotes; however, the character expected for quotes can be changed using the /Q switch. Likewise, by default the delimiter character expected to separate fields is a comma but it can be changed using the /D switch. Multiple ranges can be specified but they must be in ascending order. For example, 1 3 7 10 16 16 would result in the quoting of lines 1-3, 7-10 and 16. As seen from this example, single lines can be quoted only by specifying them twice; that is, as a range of one line. If the /S (set) switch is specified then the input source is not viewed as a single continuum of lines 1-n, but rather, as multiple, repeating groups of lines; i.e., 1-<no of sets>, 1-<no of sets>, 1-<no of sets>... In such case, all specified ranges of lines are quoted within each group. The /O (option) switch determines how quoting is performed. It can be specified as one of three values:

0: A line in range is quoted unconditionally (default).
1: A line in range is quoted only if it contains embedded quotes or delimiters.
2: A line in range is quoted only if it contains one or more characters other than “0”-“9”.

The /U switch forces the QuoteLines filter to work in reverse. That is, it is used to unquote all lines that fall within the specified range(s). It first removes any outer quotes from each line and then restores embedded quotes. For an example that illustrates use of the /U switch, see the ParseCSV filter.

The following example illustrates use of the QuoteLines filter to translate fixed-width data to quoted, comma-delimited data using option 0. Given the following fixed-width data representing zip code, name, birth date, sex, account balance and account number...

```
54667  "Tiny" Tim   05/05/1980  M   20456.45   2347281749
87896  John Doe     01/15/1960  M   234888.56  6482610982
98798  Luther, Lex  05/26/1940  M   9834.01    4566281737
67491  Betty Boop   05/26/1932  F   34772.01   1902630722
----+----1----+----2----+----3----+----4----+----5----+----6
```

the pipe...

```
SplitLines 8 21 33 37 48
TrimLinesRight
QuoteLines 1 4 6 6 /s6
AppendStr ','
JoinLines 6
StripChars 1
```

produces:

```
"54667","""Tiny"" Tim","05/05/1980","M",20456.45,"2347281749"
"87896","John Doe","01/15/1960","M",234888.56,"6482610982"
"98798","Luther, Lex","05/26/1940","M",9834.01,"4566281737"
"67491","Betty Boop","05/26/1932","F",34772.01,"1902630722"
----+----1----+----2----+----3----+----4----+----5----+----6
```

Notice that every column that is in range is quoted unconditionally as per option 0. Here, we omitted the 5th column because we knew that it represented a numeric value.

Option 2 is useful when you want just those columns that contain characters other than “0”-“9” to be quoted. Specifying the QuoteLines filter using option 2...

```
SplitLines 8 21 33 37 48
TrimLinesRight
QuoteLines 1 6 /s6 /o2
AppendStr ','
JoinLines 6
StripChars 1
```

gives us this output:

```
54667,"""Tiny"" Tim","05/05/1980","M","20456.45",2347281749
87896,"John Doe","01/15/1960","M","234888.56",6482610982
98798,"Luther, Lex","05/26/1940","M","9834.01",4566281737
67491,"Betty Boop","05/26/1932","F","34772.01",1902630722
----+----1----+----2----+----3----+----4----+----5----+----6
```

Note: Since the QuoteLines filter also works in-reverse (that is, it can unquote lines), you could use it to unquote any exceptional columns after having first quoted *all* columns using option 0.

For an example in which quoted, comma-delimited data is translated to fixed-width, see the ParseCSV filter.

ReorderColumns

syntax: ReorderColumns <char pos> <char pos> [<char pos> <char pos>...] /C /P<char pos>
summary: re-arranges the column order of each line

The ReorderColumns filter re-arranges the column order of each line, allowing columns of data to be moved about, deleted or duplicated. Multiple character position pairs--each indicating a column of text to include in the output--can be specified in any order. The result is an aggregate of all such columns of text. If the /C switch is given then the second number in each pair is interpreted as a character *count* rather than as an ending character position. This can be helpful if you know beforehand how wide each column is. If the /P switch is specified then the resulting aggregate is inserted back into each line at the given character position. This feature provides an alternative means of re-arranging the column order of the text.

Given the following text:

```
06/02/2002  02:39p  <dir>  Documents and Settings  Archived
10/28/2002  10:16a  <dir>  Outlook Mail Messages   Archived
05/12/2003  05:52p  <dir>  Program Files           Archived
04/14/2003  08:36a  <dir>  Trash                   Archived
04/01/2003  01:55p  <dir>  WINNT                   Archived
----+----1----+----2----+----3----+----4----+----5----+----6
```

Suppose you were only interested in viewing the description followed by the date and time. The following pipe...

```
ReorderColumns 28 50 1 11 13 18
```

will output just those three fields:

```
Documents and Settings 06/02/2002 02:39p
Outlook Mail Messages  10/28/2002 10:16a
Program Files          05/12/2003 05:52p
Trash                  04/14/2003 08:36a
WINNT                  04/01/2003 01:55p
----+----1----+----2----+----3----+----4----+----5----+----6
```

ReplStr

syntax: ReplStr <find string> <replace string> [<find string> <replace string>...] /B<char pos> /D<delimiter> /E<char pos> /I /P<place holder> /R
summary: replaces character strings found in the text

The ReplStr filter replaces each occurrence of <find string> found within each line with <replace string>. Multiple string replacements can be performed per call. If the /I switch is specified, the find string is searched without regard to its case. If the /R switch is specified the find string is treated as a regular expression pattern.

In addition to performing text replacements in which both the find and replace strings are specified as arguments, the ReplStr filter can also perform text replacements on an input line using a replace string that *originates* from the input line itself. If a /B switch is given, text (for use as a replace string) will be extracted from the input line at the (beginning) character position specified. This extracted text's ending position in the input line can be specified using the /E switch. As an alternative, you can omit the /E switch and by default, the text in the input line must be terminated (delimited) by a blank character. This delimiter character can be changed by specifying a /D switch.

Note: The <delimiter> is a single character and not a string; therefore, only the first character of a specified delimiter will be used.

As text is extracted from the input line itself and used as a replace string when a /B switch is specified, you might wonder what use is the replace string argument in this case? The answer is that it acts as a template to describe how the extracted text is formatted during the text replacement. This template should contain a placeholder character ("%" is used by default) and this placeholder gets replaced by the extracted text. If needed, you can change the placeholder character using the /P switch. For more clarification of the replace string argument used as a template, see the last example below.

As a first example to illustrate the basic functionality of the ReplStr filter, given the following text...

```
PipeWrench rocks
```

the following pipe...

```
ReplStr 'rocks' 'ROCKS!'
```

produces:

```
PipeWrench ROCKS!
```

Because it's possible to replace a character string with one that contains an end-of-line, ("\e" being a platform independent end-of-line used by PipeWrench) you can insert lines into the output stream using the ReplStr filter. For example, given the following text...

```
the,quick,brown,fox
```

the pipe...

```
ReplStr ',' '\e'
```

outputs:

```
the
quick
brown
fox
```

Although, lines can be inserted into the text stream by inserting end-of-lines, it's not possible to replace end-of-lines since they cannot actually be found in an input line of text. Fortunately, there's a way around this by using the JoinLines filter. This technique can be used to replace not only the end-of-lines, but also text that is adjacent to them.

For example, the following pipe uppercases any occurrence of the word "the" that is located at the end of a line:

```
AppendStr '<eol>'
JoinLines
ReplStr 'the<eol>' 'THE<eol>' '<eol>' '\e'
```

Given the text...

```
you can use the replstr filter
to replace end-of-lines with the
help of the joinlines filter
```

the above pipe will output the following:

```
you can use the replstr filter
to replace end-of-lines with THE
help of the joinlines filter
```

As you can see, only one "the" is uppercased but the other two are left unchanged as they are not at the end of a line.

Note: This is not how you would typically approach the problem of replacing strings at the end of lines. I show this only to illustrate that end-of-lines themselves can be referenced in a find string and thus be replaced or deleted (in fact, this is commonly done in PipeWrench). In actual practice, the above would be done using a regular expression:

```
ReplStr 'the$' 'THE' /r
```

As noted above, the ReplStr filter can perform text replacements using a replace string that originates from the input line itself. This feature is exploited in the "Create an Index" demo that is accessible via PipeWrench's About menu. Of relevance to this discussion are the following lines, including comments, from the middle of the pipe (Create An Index.pip):

```
; Distribute the page number across each line
; to every word on that line:

ReplStr ' ' ':% ' /b1 /d':'

; Remove the page number and ":" at the head of each line:

DelChars 1 5
```

Format-wise, the text that is input to the ReplStr filter at this point in the pipe looks like the following (the number of words is reduced here for illustration purposes):

```
0001:Joseph Priestley March Old Style February
0002:Early life and education Priestley was
0003:Priestley later wrote that the book
0004:Warrington Academy In Priestley moved to
```

Note that each of these lines ends with a space.

When run through the above two filters, the 4-digit page number at the head of each line is "distributed" across the line to each word and then removed from the head of the line:

```
Joseph:0001 Priestley:0001 March:0001 Old:0001 Style:0001 February:0001
Early:0002 life:0002 and:0002 education:0002 Priestley:0002 was:0002
Priestley:0003 later:0003 wrote:0003 that:0003 the:0003 book:0003
Warrington:0004 Academy:0004 In:0004 Priestley:0004 moved:0004 to:0004
```

In the actual demo, (Create An Index.pip) this resulting text is then further processed to form the word index, (which is beyond the scope of this example).

As noted earlier, when the ReplStr filter is used with the /B switch, the replace string argument is treated as a template. Here, in this example, the template is specified as ":% ". Again, the "%" is a placeholder that will be replaced by the text that gets extracted from the input line. The result of this replacement is then used to replace any blanks that are found in the input line. On the first line for example, the text "0001" replaces the "%" in the template resulting in ":0001 ". This resulting string is then used to replace all blanks in the first line of the input text. The remaining input lines are handled accordingly.

ReverseChars

syntax: ReverseChars [<begin char pos> <end char pos>]
summary: reverses the characters in each line of the input text

The ReverseChars filter reverses the characters in each line of the input text. If a range of character positions is specified, then just those characters will be reversed. The ReverseChars filter is mostly useful as a pre-filter for other filters. For example, given the following text...

```
brake pads 12.50
brake drums 32.90
spark plug wires 06.75
radiator hose 05.49
```

suppose you wanted to insert a dollar sign, “$” ahead of the cost for each item. The InsStr filter alone can’t help here because the character position of the cost varies from line to line. However, if the characters on each line were reversed, the dollar sign could then be inserted at column 6 for each line. The following pipe does exactly this and then it reverses the resulting text a second time in order to return it back to it’s initial state:

```
ReverseChars
InsStr 6 '$'
ReverseChars
```

The result of its execution on the above text is as follows:

```
brake pads $12.50
brake drums $32.90
spark plug wires $06.75
radiator hose $05.49
```

Note: The particular problem presented here is merely for demonstration. Were this problem actually encountered, you’d likely use the ReplStr filter and a regular expression to solve it.

As another example, supposed we had the following list of names...

```
John
Susan
Paul
George
```

and for whatever reason, we wanted to vertically “flip” them. Though this is more easily and more efficiently done using the InsLineNo filter (as seen in the example for that filter), it can also be done using the ReverseChars filter as seen here:

```
ReverseChars
AppendStr '<eol>'
JoinLines
StripChars 1
ReverseChars
ReplStr '<eol>' '\e'
```

Execution of this pipe results in the desired output:

```
George
Paul
Susan
John
```

RightChars

syntax: RightChars <no of chars>
summary: returns the given number of characters from the end of each line of text

The RightChars filter outputs the specified right-most characters from each input line of text.

Given the text...

```
Table, 123.45
Chair, 061.22
----+----1----+----2
```

the pipe...

```
RightChars 6
```

gives:

```
123.45
061.22
----+----1----+----2
```

See also *LeftChars*

RotCharsLeft

syntax: RotCharsLeft <no of chars>
summary: rotates characters left given no of places

The RotCharsLeft filter rotates the characters in each line the specified number of characters to the left.

For an example showing the use of RotCharsLeft, see the RotCharsToStr filter.

See also *RotCharsRight, RotCharsToStr*

RotCharsRight

syntax: RotCharsRight <no of chars>
summary: rotates characters right given no of places

The RotCharsRight filter rotates the characters in each line the specified number of characters to the right.

See also *RotCharsLeft, RotCharsToStr*

RotCharsToStr

syntax: RotCharsToStr <string> /I /N<count> /R
summary: rotates each line until given string is at its beginning

The RotCharsToStr filter rotates the characters in each line left until the specified string is located at the beginning of the line. If the desired string is already located at the beginning of a line, no rotation takes place. If a count is specified using the /N switch, that number of rotate-to-string operations is carried out. This enables you to target a particular occurrence of a string that exists multiple times in a line being rotated. If the /I switch is specified, the string is matched regardless of case. If the /R switch is specified, the string is treated as a regular expression pattern.

As an example, suppose you had data consisting of 4 fields per line in which you wanted to insert a "$" ahead of the third field (cost):

```
64339; Table, w/ glass top; 123.45; Furniture Dept.
94732; Chair, high back w/ cushion; 61.22; Furniture Dept.
```

You could use the following pipe:

```
AppendStr '<eol>'
RotCharsToStr ';' /n2
InsStr 3 '$'
RotCharsToStr '<eol>'
DelChars 1 5
```

What this pipe does is, it first marks the ending location for each line using the marker string, "<eol>" and then it rotates to the head of each line, the second occurrence of a semicolon. This places the desired point of insertion at the known character position, 3. It is here that the "$" is then inserted. Afterwards, the resulting lines are restored back to their original rotation by rotating to the marker string that was originally appended, and deleting it.

The pipe's execution results in the desired change to each line:

```
64339; Table, w/ glass top; $123.45; Furniture Dept.
94732; Chair, high back w/ cushion; $61.22; Furniture Dept.
```

One use of the RotCharsToStr filter that isn't immediately obvious is to have it rotate not just each line individually but the *entire* input text. This is an extremely powerful editing technique. For an in-depth example, see the demo "Edit HTML Files". What follows is a more simplistic and concise example.

Given the following lines of text:

```
The RotCharsToStr filter
can be used to rotate
the lines in the text.
```

Suppose you wanted to rotate them upward so that the second line ended up at the top. The following pipe makes use of the RotCharsToStr filter to accomplish this:

```
AppendStr '<eol>'
JoinLines
RotCharsToStr '<eol>'
RotCharsLeft 5
StripChars 5
ReplStr '<eol>' '\e'
```

Its execution results in the following:

```
can be used to rotate
the lines in the text.
The RotCharsToStr filter
```

See also *RotCharsLeft*, *RotCharsRight*

SetDebugOn

syntax: SetDebugOn
summary: configures pipe debugging on

The SetDebugOn command turns debugging on for all subsequent filters until the SetDebugOff command is encountered.

See also *SetDebugOff*

SetDebugOff

syntax: SetDebugOff
summary: configures pipe debugging off

The SetDebugOff command terminates filter debugging that was initiated prior by the SetDebugOn command.

See also *SetDebugOn*

ShiftChars

syntax: ShiftChars <char pos> <string> [<char pos> <string>...] /I /R
summary: shifts text into specified character positions

The ShiftChars filter shifts text into specified character positions, (deleting any text that is shifted over). It will shift the text either to the left or to the right as necessary. The first parameter is the character position to shift the text to and the second parameter specifies the string to be shifted. Any number of position/string pairs can be specified but the character positions must be in ascending order. If the /I switch is specified the case of the strings is ignored. If the /R switch is specified the strings are treated as a regular expression patterns.

The ShiftChars filter is useful for forcing text into a specific column position so that subsequent filters can operate based on that assumption. For example, given the following data delimited by commas...

```
John L. Doe,42,Miami,FL
Frank N. Stein,20,Los Angeles,CA
Norma Lee,16,Austin,TX
----+----1----+----2----+----3----+----4
```

the following pipe...

```
ShiftChars 15 ',' 18 ',' 30 ','
DelChars 15 1 18 1 30 1
```

will output the data fixed-width:

```
John L. Doe   42Miami      FL
Frank N. Stein20Los AngelesCA
Norma Lee     16Austin     TX
----+----1----+----2----+----3----+----4
```

Here, the same string, (a comma) is specified repeatedly but each specified string could be different.

Note: The above is not a particularly good way to transform CSV data to fixed-width. This is partly because you don't necessarily know beforehand what column to shift the commas to (that will accommodate the widest field for that column). The preferred approach would be to use the ParseCSV filter along with the PadLinesRight and JoineLines filters. See ParseCSV for an example.

Another use for ShiftChars is to simply delete text at some given character position up to a given string (a function that the DelCharsToStr also performs). For example, suppose you wanted only the third and last fields of the following lines of data:

```
2010/08/15, 12:15:48.030, 983900002, A, 34.45.22.65, Driver Failure
2010/08/15, 12:16:01.473, 0872303, A, 34.45.22.65, Bad Drive at 45453:883
2010/08/15, 12:16:02.229, 740900002, A, 34.45.22.65, System Configured
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
```

This can be achieved using ShiftChars by making use of the comma characters. The following pipe...

```
ShiftChars 1 ',' 1 ',' 13 ',' 13 ',' 13 ','
DelChars 1 2
ReplStr '\\s+,' ',' /r
```

delivers the goods:

```
983900002, Driver Failure
0872303, Bad Drive at 45453:883
740900002, System Configured
----+----1----+----2----+----3----+----4
```

The ShiftChars filter shifts the first two commas, one after the other, into character position 1. This effectively wipes out the first two fields of data. Then, it shifts the next three commas into position 13, which effectively removes fields 4 and 5. The DelChars filter then removes the comma and space from the head of each line. And finally, the ReplStr filter uses a regular expression to remove the spaces in front of the remaining commas.

Note: When a string is referenced multiple times by the ShiftChars filter, each reference refers to the next “unshifted” occurrence of that string found in the input text. Once an occurrence of that string in the input text has been referenced it can no longer be referenced again by the same ShiftChars filter.

See also *DelCharsToStr*

SortLines

syntax: SortLines /P<char pos> /R
summary: sorts the text

The SortLines filter outputs the text sorted. The /P switch can be used to specify which character position to sort the text on. By default, the text is sorted on character position 1 (that is, the sort is performed on the entire line of text). The /R switch causes the input text to be sorted in reverse order.

Given the text...

```
the
quick brown
fox
jumped
over the lazy
dog and
the lazy
dog
didn't even bark
```

the following pipe parses each word onto a separate line, inserts a line number ahead of each word and then sorts the text based on the words, not the line numbers:

```
ParseWords
InsStr 1 ' '
InsLineNo /w2 /z
SortLines /p4
```

Its execution results in the following:

```
10 and
16 bark
03 brown
14 didn't
09 dog
13 dog
15 even
04 fox
05 jumped
12 lazy
08 lazy
06 over
02 quick
11 the
01 the
07 the
```

For a more practical example, see the IsolateLines filter.

SpliceFile

syntax: SpliceFile <file name> /D<delimiter> /M
summary: combines text from a text file to the input text

The SpliceFile filter extends each line of the input text by appending to it its corresponding line from the splice file. In other words, the two text sources are "spliced" together. If either text source has more lines of text than the other, the extra lines are simply appended to the output and therefore, the input text and the splice file should contain the same number of lines of text. If a delimiter string is specified using the /D switch, it will be appended to each line of the input text ahead of the spliced text. If the /M (merge) switch is specified the text from the splice file is added to the end of the input text rather than spliced line-by-line. Note that the <file name> parameter must be a single-quoted string as it can contain blanks.

Given the following input text...

```
test.txt
rev.txt
list1.txt
```

and the following text in list2.txt...

```
05-26-99  5:19p
05-26-99  5:22p
05-26-99  5:30p
```

the pipe...

```
PadLinesRight ' '
AppendStr ' '
SpliceFile '/home/user/list2.txt'
```

will output the following:

```
test.txt  05-26-99  5:19p
rev.txt   05-26-99  5:22p
list1.txt 05-26-99  5:30p
```

SplitLines

syntax: SplitLines <char pos> [<char pos>...]
summary: splits each line at the given character position(s)

The SplitLines filter splits each line at the given character position(s). Multiple character positions can be specified where splitting is to occur but they must be in ascending order.

Suppose for example that you wanted to reform the following list of items so that they formed four columns instead of three:

```
01 ONE      02 TWO      03 THREE
04 FOUR     05 FIVE     06 SIX
07 SEVEN    08 EIGHT    09 NINE
10 TEN      11 ELEVEN   12 TWELVE
13 THIRTEEN 14 FOURTEEN 15 FIFTEEN
----+----1----+----2----+----3----+----4
```

To do this, you would first need to get each number and its associated comment onto a separate line by itself. One way to do this is using the SplitLines filter:

```
SplitLines 13 25
```

Executing this pipe results in a single “column” of data:

```
01 ONE
02 TWO
03 THREE
04 FOUR
05 FIVE
06 SIX
07 SEVEN
08 EIGHT
...
```

Adding four more filters, we now have the completed pipe:

```
SplitLines 13 25
TrimLines
PadLinesRight ' ' /s4
AppendStr ' '
JoinLines 4
```

The TrimLines filter simply removes the trailing spaces from each line that remained after splitting the lines. The PadLinesRight filter re-pads the lines in preparation for outputting as four columns. The AppendStr filter separates each column output with a single space. Finally, the JoinLines filter recombines the lines back into tabular form.

The pipe's execution results in the desired four columns of data:

```
01 ONE      02 TWO      03 THREE   04 FOUR
05 FIVE     06 SIX      07 SEVEN   08 EIGHT
09 NINE     10 TEN      11 ELEVEN  12 TWELVE
13 THIRTEEN 14 FOURTEEN 15 FIFTEEN
----+----1----+----2----+----3----+----4----+----5
```

StripChars

syntax: StripChars <no of chars>
summary: deletes a given number of characters from the end of each line

The StripChars filter deletes a given number of characters from the end of each line. For an example of its use, see the JoinLines filter.

See also *DelChars*

SubValues

syntax: SubValues <char pos> <char pos> /I<ins char pos> /W<width> /D<decimals> /S
summary: subtracts two numbers found on each line of text

The SubValues filter subtracts two numbers found in each input line and outputs the result of the subtraction per each line. The two required parameters specify at what character positions the numbers involved in the subtraction are located and in each case must point at the leftmost digit of the number or at the white space prior to it. If the /I switch is specified, the result of the subtraction is inserted back into the line of text at the specified character position rather than simply being output alone, one per line. The /W switch determines the width of the resulting numeric values and the /D switch determines the number of decimal places displayed after the decimal point. If /S is specified, the resulting values will be output in scientific notation.

Given input text containing the two columns of numbers...

```
14       8
10       2.75
12.5     5.5
----+----1----+----2----+----3
```

the pipe...

```
SubValues 1 10
```

outputs the result of each subtraction:

```
  6.00
  7.25
  7.00
----+----1----+----2----+----3
```

Adding the /I switch...

```
SubValues 1 10 /i20
```

causes the result of the subtraction to be inserted back into each line, in this case at character position 20:

```
14        8           6.00
10        2.75        7.25
12.5      5.5         7.00
----+----1----+----2----+----3
```

TopLines

syntax: TopLines <no of lines>
summary: outputs the given number of lines from the beginning of the input text

The TopLines filter outputs the given number of lines from the top of the input text. It can be used in conjunction with the BottomLines filter to return a range of lines from the middle of the input text. See the BottomLines filter for an example of how this is done. A range of lines can also be output using LinesByPos.

See also *BottomLines*, *LinesByPos*

TotalColumns

syntax: TotalColumns <char pos> [<char pos>...] /W<width> /D<decimals> /A /S
summary: totals columns of numeric values

For each character position specified, the TotalColumns filter totals the numeric values found on all input lines at that character position. A specified character position does not have to point directly at a value's first digit, but can point to the white space ahead of the value as well. The output from the TotalColumns filter is a single line of text that contains the totals of all columns specified. The totals are written to this output line at the character positions specified. The /W switch determines the width of each numeric total and the /D switch determines the number of decimal places displayed after the decimal point. If the /A switch is specified the line of totals is output preceded by the input text lines themselves containing the values being totaled. The line of totals is thus appended to the end of the input text. If the /S switch is specified the totals will be formatted in scientific notation.

Given the following input text...

```
john     20.00      12.24
fred     52.81       6.42
lisa      8.28      19.06
thomas   20.62     103.57
tom      36.87      72.80
----+----1----+----2----+----3----+----4
```

the pipe...

```
TotalColumns 9 20
```

outputs:

```
        138.58     214.09
----+----1----+----2----+----3----+----4
```

If the decimal points of all the values in a column are lined-up as they are in this example, you generally want the resulting total's decimal point to line-up as well. The location of a column's total depends on both its starting character position specified and on the setting of the numeric width, (which can be overridden by using the /W switch). In this example the width is assumed to be the standard default of 6 therefore to have the resulting totals (their decimal points) line up properly with their respective columns, the first total's character position must be specified here as 9 instead of 10.

In general, a total's starting character position desired is determined according to the formula:

<Total's Starting Char Pos> = <Column's Right-most Char Pos> - <Numeric Width> + 1

To create reports that total both horizontally and vertically, you can use the TotalColumns filter in conjunction with the AddValues filter. Here, we use the /A (append) switch with the TotalColumns filter:

```
AddValues 10 20 /i30
TotalColumns 9 20 30 /a
```

When run, this pipe produces the following output:

```
john     20.00      12.24     32.24
fred     52.81       6.42     59.23
lisa      8.28      19.06     27.34
thomas   20.62     103.57    124.19
tom      36.87      72.80    109.67
        138.58     214.09    352.67
----+----1----+----2----+----3----+----4
```

TrimLines

syntax: TrimLines
summary: removes leading and trailing white space from each line of text

The TrimLines filter simply combines both the TrimLinesLeft and TrimLinesRight filters into one. It removes both leading and trailing white space characters from each line of text.

See also *TrimLinesLeft*, *TrimLinesRight*

TrimLinesLeft

syntax: TrimLinesLeft
summary: removes leading white space from each line of text

The TrimLinesLeft filter removes all leading blanks and tabs from the front of each line.

See also *TrimLinesRight*, *TrimLines*

TrimLinesRight

syntax: TrimLinesRight
summary: removes trailing white space from each line of text

The TrimLinesRight filter removes all trailing blanks and tabs from the end of each line.

See also *TrimLinesLeft*, *TrimLines*

UpperCase

syntax: UpperCase
summary: converts lowercase characters to uppercase

The UpperCase filter converts lowercase characters to uppercase.

See also *LowerCase*

WrapText

syntax: WrapText <char pos> /B<break chars> /J<%jaggedness>
summary: wraps text at given character position

The WrapText filter wraps the text at the given character position. By default, lines are broken at spaces only. Use the /B switch to provide a string containing additional break characters. For example, the string "-" would cause hyphens to be considered as break characters in addition to spaces.

By default, break characters will be used that are within 25% of the end of a line (thus allowing lines to be 25% jagged). You can change this by specifying a different percentage value using the /J switch.

To illustrate the WrapText filter, suppose you had the following text...

```
If you allow 100% jaggedness, then the algorithm will favor breaking the
line even at a blank (or break char.) that it finds located near the
text's left margin, thus possibly resulting in "loose" lines that contain
just a word or two.  If set to 0%, no blank or break character found will
be used as a break point and instead, the line will simply be broken at
the margin (hence 0% jaggedness).
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
```

and you wanted it to be wrapped at character position 30. This can be accomplished using the following pipe:

```
TrimLines
AppendStr ' '
JoinLines
WrapText 30
```

Since the input text is already on multiple lines, the first three filters are needed to transform it onto a single line. Then, the WrapText filter, wraps the text as desired:

```
If you allow 100% jaggedness,
then the algorithm will favor
breaking the line even at a
blank (or break char.) that
it finds located near the
text's left margin, thus
possibly resulting in "loose"
lines that contain just a
word or two.  If set to 0%,
no blank or break character
found will be used as a break
point and instead, the line
will simply be broken at the
margin (hence 0% jaggedness).
----+----1----+----2----+----3----+----4
```

Appendix B: Adding A New Filter to PipeWrench

With programming languages in general, there is usually more than one solution that can be applied to a particular programming problem. Likewise, with PipeWrench, a typical text processing problem might be solvable via many different filter combinations; but occasionally, you may find it difficult to arrive at a solution to a problem using PipeWrench. When this occurs, it may be that a new filter needs to be developed and added into PipeWrench. Fortunately, PipeWrench allows for this via the use of plugins. Included with PipeWrench is a sample plugin surprisingly called, “SamplePlugin”. This plugin contains a single filter called “SampleFilter”. SampleFilter is, in actuality, a copy of the PipeWrench core filter, *InsStr*.

Following is the code implementing SampleFilter (in SamplePlugin.cs):

```
using System;
using Firefly.PipeWrench;

namespace SamplePlugin
{
   public sealed class SampleFilter : FilterPlugin
   {
      public override void Execute()
      {
         int charPos;
         string charStr;

         Open();

         try
         {
            while (!EndOfText)
            {
               int offset = 0;
               int prevPos = 0;

               // Read a line of the source text:

               string text = ReadLine();

               // Insert each string argument into it:

               for (int j = 0; j < CmdLine.ArgCount / 2; j++)
               {
                  charPos = (int) CmdLine.GetArg((j*2)).Value + offset;
                  CheckIntRange(charPos, 1, int.MaxValue, "Character position",
                  CmdLine.GetArg((j*2)).CharPos);

                  if (charPos > prevPos)
```

```
                {
                   prevPos = charPos;
                   charStr = (string) CmdLine.GetArg((j*2)+1).Value;
                   if (charStr == string.Empty) ThrowException("String cannot be empty.",
                   CmdLine.GetArg((j*2)+1).CharPos);

                   while (text.Length < charPos - 1)
                   {
                      text += ' ';
                   }

                   text = text.Insert(charPos-1, charStr);
                   offset += charStr.Length;
                }
                else
                {
                   // Oops!

                   ThrowException("Character position arguments must be in ascending order.",
                   CmdLine.GetArg((j*2)).CharPos);
                }
             }

             // Write the edited line to the output file:

             WriteText(text);
          }
       }

       finally
       {
          Close();
       }
    }

    public SampleFilter(IFilter host) : base(host)
    {
       Template = "n s [n s...]";
    }
  }
}
```

SampleFilter can be used as a basis for creating your own filters. Starting with its code, I'll step you through the process of creating a new "CountWords" filter that will output the number of words contained in its input text. As this tutorial is intended only to demonstrate how to create a PipeWrench filter, it will be carried out from a Linux "live CD/DVD" so that no permanent changes need be made to your system.[3] An added bonus of carrying out these instructions in a "live" environment is that it doesn't matter whether your computer is running Linux or Windows—the instructions are the same either way. If you prefer, a USB stick can be used in place of a CD/DVD if your system has no optical drive.

Creating a CountWords Filter

1. The first step in the tutorial is to go to the Linux Mint downloads page (http://www.linuxmint.com/download.php) and download the appropriate "live" image (ISO) for your hardware. The current Mint release is 17 (codenamed "Qiana"). It is an "LTS" release that will be supported until April of 2019. The instructions given here ought to apply regardless of which particular desktop environment you choose, Cinnamon or MATE (pronounced "Ma-Tay").

2. Once downloaded, burn the ISO image to a DVD (or a CD if it will fit). Again, you may prefer to write the image to a USB stick and boot from that instead.

3. Next, boot your system with the live medium. You may have to set your computer's bios to allow booting from the chosen medium for this to work.

4. Once booted into the live environment, you need to ensure that you have access to the internet. If your computer is connected via ethernet, you should be good to go, but you might test the connection in the internet browser just to be sure. If your system is set up to use Wi-Fi, click on the networking applet on the panel to connect to your Wi-Fi router (you'll be asked to enter your password).

5. Add the PPA for PipeWrench to your system's software sources. From the desktop environment's main menu, run the "Software Sources" application. Click the "PPAs" tab and then "Add a new PPA". In the resulting text control, enter "ppa:bwb-s/pipewrench" and click "Ok". It'll display a description for PipeWrench. Click "Ok". Afterwards you'll see the new PPA listed.[4] Close the dialog window.

6. Now, from the desktop environment's main menu, open the "terminal" application. In it, run the command, "sudo apt-get update". This will update the package cache for the running (live) system so that it has the latest package information available to it when we install software.

7. Install PipeWrench. Again in the terminal, run the command, "sudo apt-get install pipewrench".

---

3 Working from a live medium is a convenient and generally safe means of testing software on your system without your having to worry about making permanent changes to it. When your system boots from a live medium, it runs completely in memory. Even the live environment's writable disk space is all in memory, so you can run programs, install software, do whatever and unless you set out to permanently alter your system, no changes will be made to it. About the only way you could do actual harm to your computer while running from a live CD is if you accidentally clicked the "install" icon on the desktop (and accidentally follow the installation prompts). Granted, you could accidentally mount your computer's real disk drive and accidentally delete files from it, but seriously...

4 The page on launchpad where PipeWrench can be found is: https://launchpad.net/~bwb-s.

8. Install the Monodevelop IDE. In the terminal run, “sudo apt-get install mono-complete monodevelop”. Normally you wouldn’t need to install “mono-complete” along with Monodevelop but due to a bug in Mint 17, this is necessary.

9. Now, from the desktop environment’s main menu, open both PipeWrench and Monodevelop.

10. In Monodevelop create a new C# *library* project. Call it “MyPlugin” and save its project folder onto your desktop.

11. Replace the “skeleton” class (MyClass) inside of the new library project with the code from the “SampleFilter” class that’s found inside of SamplePlugin.cs (see the above listing) and rename it to “CountWords” (be sure to also rename the constructor method near the bottom). Optionally, you could rename the file itself (MyClass.cs) to reflect this new name. To do this, simply right-click the class name (“CountWords”) on line 6 or thereabouts and choose “Refactor” and then “Rename to CountWords.cs”.

12. In Monodevelop, you need to “edit” the references for the project to include a reference to the file, “Shared.dll” that gets copied to your computer when PipeWrench is installed. You can do this by right-clicking “references” beneath your project, “MyPlugin” which is found in the “Solutions” pane docked on the left side of Monodevelop’s main window. Choose the “.NET Assembly” option and then browse to the Shared.dll file on your system (it should be located at /usr/lib/pipewrench/Shared.dll).

13. Next, be sure that “Release” is selected instead of “Debug” for the project’s type of build.

14. Now we’re ready to build the project. On the “Build” menu, click “Build MyPlugin”.

15. Assuming that the above build was successful, you can now install the new plugin into PipeWrench. For PipeWrench to use the new plugin, it needs to be listed in PipeWrench’s engine configuration file: ~/.pipewrench/EngineConf.xml.[5] Open this file using the desktop environment’s default text editor. You’ll notice that the sample filter is already listed under *filters*. You merely need to add the new filter after it. For now, simply copy the line for SampleFilter and change it as shown here:

    ```
    <Filter Name="MyPlugin.CountWords" Enabled="true" Icon="gtk-connect" Template="n s [n s...]" Desc="Returns count of words"
    Prompt="<charpos> <string> [<charpos> <string>...]" AssyName="MyPlugin.dll"/>
    ```

    You can leave this file open in the text editor. Just be sure to save the file after you make the changes.

16. Additionally, the plugin itself (MyPlugin.dll) must reside in PipeWrench’s plugin directory. On a Linux system this directory is “/usr/lib/pipewrench/PlugIns”. First, we need to make sure that this directory exists. In the terminal, type “sudo mkdir -p /usr/lib/pipewrench/PlugIns”. Now, we could copy the new plugin into this directory, but a better approach for our purposes is to create a symlink in this directory that references the

5 For non-Linux types, “~/” represents the Linux user's “home” directory. When running from a Linux Mint live CD, the Linux user is “mint”. Therefore, “~/” is equivalent to “/home/mint”.

actual file. That way, if we need to rebuild the plugin, we won't have to re-copy it to PipeWrench's plugin directory afterwards. So, in the terminal run the command, "sudo ln -s /home/mint/Desktop/MyPlugin/MyPlugin/bin/Release/MyPlugin.dll /usr/lib/pipewrench/PlugIns/MyPlugin.dll".

17. If PipeWrench is still running, close and re-open it. You should now see the new CountWords filter listed on the left side of the PipeWrench window.

18. Try out the new filter by typing it into the *pipe* control and then running this one-filter pipe against the default text. PipeWrench will complain that it "expected an integer parameter...". This is because our new filter is a *copy* of the sample filter which requires arguments.

19. To make the new filter behave as it should, we first need to make some minor changes to the "EngineConf.xml" file. The file should still be open in the text editor. Simply change the line for our new filter so that both its template and prompt strings are empty:

    ```
    <Filter Name="MyPlugin.CountWords" Enabled="true" Icon="gtk-connect" Template="" Desc="Returns count of words" Prompt=""
    AssyName="MyPlugin.dll"/>
    ```

    This is needed because our new filter, CountWords doesn't require any arguments.

20. Also, the new filter's code must be modified so that the filter outputs the number of words found in the input text. Make the necessary changes as per the completed code listing that follows these instructions (you can simply copy the entire listing if you'd like) and save the file.

21. Now re-build the project.

22. If PipeWrench is currently open, close and then re-open it.

23. Again, in PipeWrench add the CountWords filter to the pipe control and run the pipe against the default text. This time, the pipe should output the text, "10", indicating the number of words that were found in the input text.

Congratulations, you have created your first PipeWrench filter!

Following is the completed code implementing the new CountWords filter:

```
using System;
using Firefly.PipeWrench;

namespace MyPlugin
{
   public sealed class CountWords : FilterPlugin
   {
      public override void Execute()
      {
         Open();

         try
         {
            int total = 0;

            // Total up the words:

            while (!EndOfText)
            {
               // Read a line of the source text:

               string text = ReadLine();

               // Add the count of words found in the line to the total:

               total += text.Split(' ').Length;
            }

            // Output the total words:

            WriteText(total.ToString());
         }

         finally
         {
            Close();
         }
      }

      public CountWords(IFilter host) : base(host)
      {
         Template = "";
      }
   }
}
```

Appendix C: Regular Expressions Reference

Following are some of the more common *regex* elements that can be used to create regular expression patterns for use in PipeWrench pipes:[6]

Regex Element	Description
\t	Matches a tab character.
\e	Matches an escape character.
\a	Matches a bell character.
\v	Matches a vertical tab character.
\f	Matches a form feed character.
\	Matches the character that follows it (unless the sequence is otherwise recognized as an “escape”)
\w	Matches any word character.
\W	Matches any non-word character.
\d	Matches any decimal digit.
\D	Matches any character other than a decimal digit.
\s	Matches any whitespace character.
\S	Matches any non-whitespace character.
\nnn	Matches character represented by “nnn” expressed in octal. Must be comprised of exactly 3 digits.
\xnn	Matches character represented by “nn” expressed in hexadecimal.
\cnnnn	Matches control character represented by “nnnn”.
\unnnn	Matches unicode character represented by “nnnn”.
.	Matches any single character except newline (\n).
[*group*]	Matches any single character in the *group* of characters.
[^*group*]	Matches any single character that is not in the *group* of characters.
[*char-char*]	Matches any single character that is in the range of characters.
^	Matches the beginning of a line.
$	Matches the end of a line.
*	Matches zero or more of the previous character.
+	Matches one or more of the previous character.
?	Matches the previous element zero or one time.

6 This table includes only a subset of the *regex* elements implemented in PipeWrench and is by no means complete. As PipeWrench is a .NET application, its *regex* capabilities are based on .NET’s implementation of *regex*. For a more complete list of regex elements implemented in PipeWrench, please see Microsoft’s online documentation regarding regular expressions.

Regex Element	Description
{n}	Matches the previous element exactly *n* times.
{n,}	Matches the previous element at least *n* times.
{n,m}	Matches the previous element at least *n* times but no more than *m* times.
*?	Matches the previous element zero or more times, but as few times as possible.
+?	Matches the previous element one or more times, but as few times as possible.
??	Matches the previous element zero or one time, but as few times as possible.
{n}?	Matches the preceding element exactly n times.
{n,}?	Matches the previous element at least n times, but as few times as possible.
{n,m}?	Matches the previous element between n and m times, but as few times as possible.
|	Matches any one of multiple elements delimited by the "|" character.
(*subgroup*)	Captures the matched subgroup and assigns it a 1-based ordinal number (its relative position in the pattern) that can be referenced both later in the *match* string (as well as in a paired *replace* string as per the ReplStr filter).
\num	Matches the previously captured subgroup specified by *num* where num is the subgroup's relative 1-based position in the pattern.
$*num*	In a *replace* string (ReplStr filter), substitutes the string that is matched by the subgroup number, *num*. If num is "0", then the entire pattern match is used.
$&	Substitutes a copy of the whole match.
$`	Substitutes all the text of the input string before the match.
$'	Substitutes all the text of the input string after the match.
$+	Substitutes the last group that was captured.
$_	Substitutes the entire input string.
$$	In a replace string (ReplStr filter), substitutes a literal "$".

Appendix D: The ASCII Character Set

In the chart below, the hexadecimal ASCII value of a character is found by adding the character's row value, on the left, to it's column value given at the top. For example the ASCII value for an asterisk, "*" is 20 + A or 2A.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL