

# Advanced gawk

## IN THIS CHAPTER

Reexamining gawk

Using variables in gawk

Using structured commands

Formatting your printing

Working with functions

Chapter 19 introduced the `gawk` program and demonstrated the basics of using it to produce formatted reports from raw data files. This chapter dives more deeply into customizing `gawk` to produce reports. The `gawk` program is a full-fledged programming language, providing features that allow you to write advanced programs to manipulate data. If you are jumping into the shell script world from another programming language, you should feel right at home with `gawk`. In this chapter, you'll see how to use the `gawk` programming language to write programs to handle just about any data-formatting task you'll run into.

## Using Variables

---

One important feature of any programming language is the ability to store and recall values using variables. The `gawk` programming language supports two different types of variables:

- Built-in variables
- User-defined variables

Several built-in variables are available for you to use in `gawk`. The built-in variables contain information used in handling the data fields and records in the data file. You can also create your own variables in your `gawk` programs. The following sections walk you through how to use variables in your `gawk` programs.

## Built-in variables

The `gawk` program uses built-in variables to reference specific features within the program data. This section describes the built-in variables available for you to use in your `gawk` programs and demonstrates how to use them.

### The field and record separator variables

Chapter 19 demonstrated one type of built-in variable available in `gawk`: the *data field variables*. The data field variables allow you to reference individual data fields within a data record using a dollar sign and the numerical position of the data field in the record. Thus, to reference the first data field in the record, you use the `$1` variable. To reference the second data field, you use the `$2` variable, and so on.

Data fields are delineated by a field separator character. By default, the field separator character is a whitespace character, such as a space or a tab. Chapter 19 showed how to change the field separator character either on the command line by using the `-F` command line parameter or within the `gawk` program using the special `FS` built-in variable.

The `FS` built-in variable belongs to a group of built-in variables that control how `gawk` handles fields and records in both input data and output data. Table 22-1 lists the built-in variables contained in this group.

**TABLE 22-1    The `gawk` Data Field and Record Variables**

Variable	Description
<code>FIELDWIDTHS</code>	A space-separated list of numbers defining the exact width (in spaces) of each data field
<code>FS</code>	Input field separator character
<code>RS</code>	Input record separator character
<code>OFS</code>	Output field separator character
<code>ORS</code>	Output record separator character

The `FS` and `OFS` variables define how your `gawk` program handles data fields in the data stream. You've already seen how to use the `FS` variable to define what character separates data fields in a record. The `OFS` variable performs the same function but for the output by using the `print` command.

By default, `gawk` sets the `OFS` variable to a space, so when you use the command:

```
print $1,$2,$3
```

you see the output as:

```
field1 field2 field3
```

You can see this in the following example:

```
$ cat data1
data11,data12,data13,data14,data15
data21,data22,data23,data24,data25
data31,data32,data33,data34,data35
$ gawk 'BEGIN{FS=","} {print $1,$2,$3}' data1
data11 data12 data13
data21 data22 data23
data31 data32 data33
$
```

The `print` command automatically places the value of the `OFS` variable between each data field in the output. By setting the `OFS` variable, you can use any string to separate data fields in the output:

```
$ gawk 'BEGIN{FS=","; OFS="-"} {print $1,$2,$3}' data1
data11-data12-data13
data21-data22-data23
data31-data32-data33
$ gawk 'BEGIN{FS=","; OFS="--"} {print $1,$2,$3}' data1
data11--data12--data13
data21--data22--data23
data31--data32--data33
$ gawk 'BEGIN{FS=","; OFS="<-->"} {print $1,$2,$3}' data1
data11<-->data12<-->data13
data21<-->data22<-->data23
data31<-->data32<-->data33
$
```

The `FIELDWIDTHS` variable allows you to read records without using a field separator character. In some applications, instead of using a field separator character, data is placed in specific columns within the record. In these instances, you must set the `FIELDWIDTHS` variable to match the layout of the data in the records.

After you set the `FIELDWIDTHS` variable, gawk ignores the `FS` and calculates data fields based on the provided field width sizes. Here's an example using field widths instead of field separator characters:

```
$ cat data1b
1005.3247596.37
115-2.349194.00
05810.1298100.1
$ gawk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1,$2,$3,$4}' data1b
100 5.324 75 96.37
115 -2.34 91 94.00
058 10.12 98 100.1
$
```

The `FIELDWIDTHS` variable defines four data fields, and `gawk` parses the data record accordingly. The string of numbers in each record is split based on the defined field width values.

### CAUTION

It's important to remember that after you set the `FIELDWIDTHS` variable, those values must remain constant. This method can't accommodate variable-length data fields.

The `RS` and `ORS` variables define how your `gawk` program handles records in the data stream. By default, `gawk` sets the `RS` and `ORS` variables to the newline character. The default `RS` variable value indicates that each new line of text in the input data stream is a new record.

Sometimes, you run into situations where data fields are spread across multiple lines in the data stream. A classic example of this is data that includes an address and phone number, each on a separate line:

```
Riley Mullen
123 Main Street
Chicago, IL 60601
(312)555-1234
```

If you try to read this data using the default `FS` and `RS` variable values, `gawk` reads each line as a separate record and interprets each space in the record as a field separator. This isn't what you intended.

To solve this problem, you need to set the `FS` variable to the newline character. This indicates that each line in the data stream is a separate field and all the data on a line belongs to the data field. However, when you do that, you don't know where a new record starts.

To solve this problem, set the `RS` variable to an empty string, and leave a blank line between data records in the data stream. The `gawk` program interprets each blank line as a record separator.

The following is an example of using this technique:

```
$ cat data2
Riley Mullen
123 Main Street
Chicago, IL 60601
(312)555-1234

Frank Williams
456 Oak Street
Indianapolis, IN 46201
```

```
(317)555-9876

Haley Snell
4231 Elm Street
Detroit, MI 48201
(313)555-4938
$ gawk 'BEGIN{FS="\n"; RS=""} {print $1,$4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

Perfect! The gawk program interpreted each line in the file as a data field and the blank lines as record separators.

Data variables

Besides the field and record separator variables, gawk provides some other built-in variables to help you know what’s going on with your data and extract information from the shell environment. Table 22-2 shows the other built-in variables in gawk.

TABLE 22-2 More gawk Built-In Variables

Variable	Description
ARGC	The number of command line parameters present
ARGIND	The index in ARGV of the current file being processed
ARGV	An array of command line parameters
CONVFMT	The conversion format for numbers (see the printf statement), with a default value of %.6 g
ENVIRON	An associative array of the current shell environment variables and their values
ERRNO	The system error if an error occurs when reading or closing input files
FILENAME	The filename of the data file used for input to the gawk program
FNR	The current record number in the data file
IGNORECASE	If set to a non-zero value, ignores the case of characters in strings used in the gawk command
NF	The total number of data fields in the data file
NR	The number of input records processed
OFMT	The output format for displaying numbers, with a default of %.6 g
RLENGTH	The length of the substring matched in the match function
RSTART	The start index of the substring matched in the match function

You should recognize a few of these variables from your shell script programming. The ARGC and ARGV variables allow you to retrieve the number of command line parameters and

their values from the shell. This can be a little tricky, however, because `gawk` doesn't count the program script as part of the command line parameters:

```
$ gawk 'BEGIN{print ARGV[1]}' data1
2 data1
$
```

The `ARGC` variable indicates that two parameters are on the command line. This includes the `gawk` command and the `data1` parameter (remember that the program script doesn't count as a parameter). The `ARGV` array starts with an index of 0, which represents the command. The first array value is the first command line parameter after the `gawk` command.

### NOTE

Note that unlike shell variables, when you reference a `gawk` variable in the script, you don't add a dollar sign before the variable name.

The `ENVIRON` variable may seem a little odd to you. It uses an *associative array* to retrieve shell environment variables. An associative array uses text for the array index values instead of numeric values.

The text in the array index is the shell environment variable. The value of the array is the value of the shell environment variable. The following is an example of this:

```
$ gawk '
> BEGIN{
> print ENVIRON["HOME"]
> print ENVIRON["PATH"]
> }'
/home/rich
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
$
```

The `ENVIRON["HOME"]` variable retrieves the `HOME` environment variable value from the shell. Likewise, the `ENVIRON["PATH"]` variable retrieves the `PATH` environment variable value. You can use this technique to retrieve any environment variable value from the shell to use in your `gawk` programs.

The `FNR`, `NF`, and `NR` variables come in handy when you're trying to keep track of data fields and records in your `gawk` program. Sometimes, you're in a situation where you don't know exactly how many data fields are in a record. The `NF` variable allows you to specify the last data field in the record without having to know its position:

```
$ gawk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}' /etc/passwd
rich:/bin/bash
testy:/bin/csh
mark:/bin/bash
```

```
dan:/bin/bash
mike:/bin/bash
test:/bin/bash
$
```

The `NF` variable contains the numerical value of the last data field in the data file. You can then use it as a data field variable by placing a dollar sign in front of it.

The `FNR` and `NR` variables are similar to each other, but slightly different. The `FNR` variable contains the number of records processed in the current data file. The `NR` variable contains the total number of records processed. Let's look at a couple of examples to see this difference:

```
$ gawk 'BEGIN{FS=","}{print $1,"FNR="FNR}' data1 data1
data11 FNR=1
data21 FNR=2
data31 FNR=3
data11 FNR=1
data21 FNR=2
data31 FNR=3
$
```

In this example, the `gawk` program command line defines two input files. (It specifies the same input file twice.) The script prints the first data field value and the current value of the `FNR` variable. Notice that the `FNR` value was reset to 1 when the `gawk` program processed the second data file.

Now, let's add the `NR` variable and see what that produces:

```
$ gawk '
> BEGIN {FS=","}
> {print $1,"FNR="FNR,"NR="NR}
> END{print "There were",NR,"records processed"}' data1 data1
data11 FNR=1 NR=1
data21 FNR=2 NR=2
data31 FNR=3 NR=3
data11 FNR=1 NR=4
data21 FNR=2 NR=5
data31 FNR=3 NR=6
There were 6 records processed
$
```

The `FNR` variable value was reset when `gawk` processed the second data file, but the `NR` variable maintained its count into the second data file. The bottom line is that if you're using only one data file for input, the `FNR` and `NR` values are the same. If you're using multiple data files for input, the `FNR` value is reset for each data file, and the `NR` value keeps count throughout all the data files.

### NOTE

When using `gawk`, notice that the `gawk` script can often become larger than the rest of your shell script. In the examples in this chapter, for simplicity we just run the `gawk` scripts directly from the command line, using the multi-line feature of the shell. When you use `gawk` in a shell script, you should place different `gawk` commands on separate lines. This makes it much easier for you to read and follow, rather than trying to cram it all onto one line in the shell script. Also, if you find yourself using the same `gawk` scripts in different shell scripts, you can save the `gawk` script in a separate file and reference it using the `-f` parameter (see Chapter 19).

## User-defined variables

Just like any other self-respecting programming language, `gawk` allows you to define your own variables for use within the program code. A `gawk` user-defined variable name can be any number of letters, digits, and underscores, but it can't begin with a digit. It is also important to remember that `gawk` variable names are case sensitive.

### Assigning variables in scripts

Assigning values to variables in `gawk` programs is similar to doing so in a shell script, using an *assignment statement*:

```
$ gawk '  
> BEGIN{  
> testing="This is a test"  
> print testing  
> }'  
This is a test  
$
```

The output of the `print` statement is the current value of the `testing` variable. Like shell script variables, `gawk` variables can hold either numeric or text values:

```
$ gawk '  
> BEGIN{  
> testing="This is a test"  
> print testing  
> testing=45  
> print testing  
> }'  
This is a test  
45  
$
```

In this example, the value of the `testing` variable is changed from a text value to a numeric value.



Assignment statements can also include mathematical algorithms to handle numeric values:

```
$ gawk 'BEGIN{x=4; x= x * 2 + 3; print x}'
11
$
```

As you can see from this example, the gawk programming language includes the standard mathematical operators for processing numerical values. These can include the remainder symbol (%) and the exponentiation symbol (using either ^ or \*\*).

### Assigning variables on the command line

You can also use the gawk command line to assign values to variables for the gawk program. This allows you to set values outside of the normal code, changing values on the fly. Here's an example of using a command line variable to display a specific data field in the file:

```
$ cat script1
BEGIN{FS=","}
{print $n}
$ gawk -f script1 n=2 data1
data12
data22
data32
$ gawk -f script1 n=3 data1
data13
data23
data33
$
```

This feature allows you to change the behavior of the script without necessitating that you change the actual script code. The first example displays the second data field in the file, while the second example displays the third data field, just by setting the value of the n variable in the command line.

There's one problem with using command line parameters to define variable values. When you set the variable, the value isn't available in the BEGIN section of the code:

```
$ cat script2
BEGIN{print "The starting value is",n; FS=","}
{print $n}
$ gawk -f script2 n=3 data1
The starting value is
data13
data23
data33
$
```

You can solve this using the `-v` command line parameter. This allows you to specify variables that are set before the `BEGIN` section of code. The `-v` command line parameter must be placed before the script code in the command line:

```
$ gawk -v n=3 -f script2 data1
The starting value is 3
data13
data23
data33
$
```

Now the `n` variable contains the value set in the command line during the `BEGIN` section of code.

## Working with Arrays

---

Many programming languages provide arrays for storing multiple values in a single variable. The `gawk` programming language provides the array feature using *associative arrays*.

Associative arrays are different from numerical arrays in that the index value can be any text string. You don't have to use sequential numbers to identify data elements contained in the array. Instead, an associative array consists of a hodge-podge of strings referencing values. Each index string must be unique and uniquely identifies the data element that's assigned to it. If you're familiar with other programming languages, this is the same concept as hash maps or dictionaries.

The following sections walk you through using associative array variables in your `gawk` programs.

### Defining array variables

You can define an array variable using a standard assignment statement. Here's the format of the array variable assignment:

```
var[index] = element
```

In this example, *var* is the variable name, *index* is the associative array index value, and *element* is the data element value. Here are some examples of array variables in `gawk`:

```
capital["Illinois"] = "Springfield"
capital["Indiana"] = "Indianapolis"
capital["Ohio"] = "Columbus"
```

When you reference an array variable, you must include the index value to retrieve the appropriate data element value:

```
$ gawk 'BEGIN{
> capital["Illinois"] = "Springfield"
```

```
> print capital["Illinois"]
> }'
Springfield
$
```

When you reference the array variable, the data element value appears. This also works with numeric data element values:

```
$ gawk 'BEGIN{
> var[1] = 34
> var[2] = 3
> total = var[1] + var[2]
> print total
> }'
37
$
```

As you can see from this example, you can use array variables just as you would any other variable in the gawk program.

## Iterating through array variables

The problem with associative array variables is that you might not have any way of knowing what the index values are. Unlike numeric arrays, which use sequential numbers for index values, an associative array index can be anything.

If you need to iterate through an associate array in gawk, you can use a special format of the `for` statement:

```
for (var in array)
{
    statements
}
```

The `for` statement loops through the statements, each time assigning the variable `var` the next index value from the `array` associative array. It's important to remember that the variable is the value of the index and not the data element value. You can easily extract the data element value by using the variable as the array index:

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> var["m"] = 3
> var["u"] = 4
> for (test in var)
> {
>     print "Index:",test," - Value:",var[test]
> }
> }'
```

```
Index: u - Value: 4
Index: m - Value: 3
Index: a - Value: 1
Index: g - Value: 2
$
```

Notice that the index values aren't returned in any particular order, but they each reference the appropriate data element value. This is somewhat important to know, because you can't count on the returned values being in the same order, just that the index and data values match.

### Deleting array variables

Removing an array index from an associative array requires a special command:

```
delete array[index]
```

The `delete` command removes the associative index value and the associated data element value from the array:

```
$ gawk 'BEGIN{
> var["a"] = 1
> var["g"] = 2
> for (test in var)
> {
>     print "Index:",test," - Value:",var[test]
> }
> delete var["g"]
> print "---"
> for (test in var)
>     print "Index:",test," - Value:",var[test]
> }'
Index: a - Value: 1
Index: g - Value: 2
---
Index: a - Value: 1
$
```

After you delete an index value from the associative array, you can't retrieve it.

### Using Patterns

---

The `gawk` program supports several types of matching patterns to filter data records, in much the same way as the `sed` editor. Chapter 19 showed two special patterns in action. The `BEGIN` and `END` keywords are special patterns that execute statements before or after the data stream data has been read. Similarly, you can create other patterns to execute statements when matching data appears in the data stream.

This section demonstrates how to use matching patterns in your gawk scripts to limit what records a program script applies to.

## Regular expressions

Chapter 20 showed how to use regular expressions as matching patterns. You can use either a Basic Regular Expression (BRE) or an Extended Regular Expression (ERE) to filter which lines in the data stream the program script applies to.

When using a regular expression, the regular expression must appear before the left brace of the program script that it controls:

```
$ gawk 'BEGIN{FS=","} /11/{print $1}' data1
data11
$
```

The regular expression `/11/` matches records that contain the string `11` anywhere in the data fields. The gawk program matches the defined regular expression against all the data fields in the record, including the field separator character:

```
$ gawk 'BEGIN{FS=","} /,d/{print $1}' data1
data11
data21
data31
$
```

This example matches the comma used as the field separator in the regular expression. This is not always a good thing. It can lead to problems trying to match data specific to one data field that may also appear in another data field. If you need to match a regular expression to a specific data instance, you should use the matching operator.

## The matching operator

The *matching operator* allows you to restrict a regular expression to a specific data field in the records. The matching operator is the tilde symbol (`~`). You specify the matching operator, along with the data field variable, and the regular expression to match:

```
$1 ~ /^data/
```

The `$1` variable represents the first data field in the record. This expression filters records where the first data field starts with the text `data`. The following is an example of using the matching operator in a gawk program script:

```
$ gawk 'BEGIN{FS=","} $2 ~ /^data2/{print $0}' data1
data21,data22,data23,data24,data25
$
```

The matching operator compares the second data field with the regular expression `/^data2/`, which indicates the string starts with the text `data2`.

This is a powerful tool that is commonly used in `gawk` program scripts to search for specific data elements in a data file:

```
$ gawk -F: '$1 ~ /rich/{print $1,$NF}' /etc/passwd
rich /bin/bash
$
```

This example searches the first data field for the text `rich`. When it finds the pattern in a record, it prints the first and last data field values of the record.

You can also negate the regular expression match by using the `!` symbol:

```
$1 !~ /expression/
```

If the regular expression isn't found in the record, the program script is applied to the record data:

```
$ gawk -F: '$1 !~ /rich/{print $1,$NF}' /etc/passwd
root /bin/bash
daemon /bin/sh
bin /bin/sh
sys /bin/sh
--- output truncated ---
$
```

In this example, the `gawk` program script prints the userid and shell for all the entries in the `/etc/passwd` file that don't match the userid `rich`!

## Mathematical expressions

In addition to regular expressions, you can also use mathematical expressions in the matching pattern. This feature comes in handy when matching numerical values in data fields. For example, if you want to display all the system users who belong to the root users group (group number 0), you could use this script:

```
$ gawk -F: '$4 == 0{print $1}' /etc/passwd
root
sync
shutdown
halt
operator
$
```

The script checks for records where the fourth data field contains the value 0. On this Linux system, five user accounts belong to the root user group.

You can use any of the normal mathematical comparison expressions:

- `x == y`: Value `x` is equal to `y`.
- `x <= y`: Value `x` is less than or equal to `y`.
- `x < y`: Value `x` is less than `y`.
- `x >= y`: Value `x` is greater than or equal to `y`.
- `x > y`: Value `x` is greater than `y`.

You can also use expressions with text data, but you must be careful. Unlike regular expressions, expressions are an exact match. The data must match exactly with the pattern:

```
$ gawk -F, '$1 == "data"{print $1}' data1
$
$ gawk -F, '$1 == "data11"{print $1}' data1
data11
$
```

The first test doesn't match any records because the first data field value isn't `data` in any of the records. The second test matches one record with the value `data11`.

22

## Structured Commands

The gawk programming language supports the usual cast of structured programming commands. This section describes each of these commands and demonstrates how to use them within a gawk programming environment.

### The if statement

The gawk programming language supports the standard `if-then-else` format of the `if` statement. You must define a condition for the `if` statement to evaluate, enclosed in parentheses. If the condition evaluates to a `TRUE` condition, the statement immediately following the `if` statement is executed. If the condition evaluates to a `FALSE` condition, the statement is skipped. You can use this format:

```
if (condition)
    statement1
```

Or you can place it on one line, like this:

```
if (condition) statement1
```

Here's a simple example demonstrating this format:

```
$ cat data4
10
```

```
5
13
50
34
$ gawk '{if ($1 > 20) print $1}' data4
50
34
$
```

Not too complicated. If you need to execute multiple statements in the `if` statement, you must enclose them with braces:

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }
> }' data4
100
68
$
```

Be careful that you don't confuse the `if` statement braces with the braces used to start and stop the program script. The `gawk` program can detect missing braces and produces an error message if you mess up:

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> }' data4
gawk: cmd. line:6: }
gawk: cmd. line:6: ^ unexpected newline or end of string
$
```

The `gawk` `if` statement also supports the `else` clause, allowing you to execute one or more statements if the `if` statement condition fails. Here's an example of using the `else` clause:

```
$ gawk '{
> if ($1 > 20)
> {
>   x = $1 * 2
>   print x
> } else
> {
>   x = $1 / 2
>   print x
> }
```



```
> }}' data4
5
2.5
6.5
100
68
$
```

You can use the `else` clause on a single line, but you must use a semicolon after the `if` statement section:

```
if (condition) statement1; else statement2
```

Here's the same example using the single line format:

```
$ gawk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' data4
5
2.5
6.5
100
68
$
```

This format is more compact but can be harder to follow.

## The while statement

The `while` statement provides a basic looping feature for gawk programs. Here's the format of the `while` statement:

```
while (condition)
{
    statements
}
```

The `while` loop allows you to iterate over a set of data, checking a condition that stops the iteration. This is useful if you have multiple data values in each record that you must use in calculations:

```
$ cat data5
130 120 135
160 113 140
145 170 215
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
```

```
>     total += $i
>     i++
> }
> avg = total / 3
> print "Average:",avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$
```

The `while` statement iterates through the data fields in the record, adding each value to the `total` variable and incrementing the counter variable, `i`. When the counter value is equal to 4, the `while` condition becomes `FALSE`, and the loop terminates, dropping through to the next statement in the script. That statement calculates the average and prints the average. This process is repeated for each record in the data file.

The `gawk` programming language supports using the `break` and `continue` statements in `while` loops, allowing you to jump out of the middle of the loop:

```
$ gawk '{
> total = 0
> i = 1
> while (i < 4)
> {
>     total += $i
>     if (i == 2)
>         break
>     i++
> }
> avg = total / 2
> print "The average of the first two data elements is:",avg
> }' data5
The average of the first two data elements is: 125
The average of the first two data elements is: 136.5
The average of the first two data elements is: 157.5
$
```

The `break` statement is used to break out of the `while` loop if the value of the `i` variable is 2.

## The do-while statement

The `do-while` statement is similar to the `while` statement but performs the statements before checking the condition statement. Here's the format for the `do-while` statement:

```
do
{
```

```

    statements
} while (condition)

```

This format guarantees that the statements are executed at least one time before the condition is evaluated. This comes in handy when you need to perform statements before evaluating the condition:

```

$ gawk '{
> total = 0
> i = 1
> do
> {
>     total += $i
>     i++
> } while (total < 150)
> print total }' data5
250
160
315
$

```

The script reads the data fields from each record and totals them until the cumulative value reaches 150. If the first data field is over 150 (as seen in the second record), the script is guaranteed to read at least the first data field before evaluating the condition.

## The for statement

The `for` statement is a common method used in many programming languages for looping. The gawk programming language supports the C-style of `for` loops:

```

for( variable assignment; condition; iteration process)

```

This helps simplify the loop by combining several functions in one statement:

```

$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>     total += $i
> }
> avg = total / 3
> print "Average:", avg
> }' data5
Average: 128.333
Average: 137.667
Average: 176.667
$

```

By defining the iteration counter in the `for` loop, you don't have to worry about incrementing it yourself as you did when using the `while` statement.

# Formatted Printing

You may have noticed that the `print` statement doesn't exactly give you much control over how `gawk` displays your data. About all you can do is control the output field separator character (OFS). If you're creating detailed reports, often you need to place data in a specific format and location.

The solution is to use the formatted printing command, called `printf`. If you're familiar with C programming, the `printf` command in `gawk` performs the same way, allowing you to specify detailed instructions on how to display data.

Here's the format of the `printf` command:

```
printf "format string", var1, var2 . . .
```

The *format string* is the key to the formatted output. It specifies exactly how the formatted output should appear, using both text elements and *format specifiers*. A format specifier is a special code that indicates what type of variable is displayed and how to display it. The `gawk` program uses each format specifier as a placeholder for each variable listed in the command. The first format specifier matches the first variable listed, the second matches the second variable, and so on.

The format specifiers use the following format:

```
%[modifier] control-letter
```

In this example, *control-letter* is a one-character code that indicates what type of data value will be displayed, and *modifier* defines an optional formatting feature.

Table 22-3 lists the control letters that can be used in the format specifier.

**TABLE 22-3    Format Specifier Control Letters**

Control Letter	Description
c	Displays a number as an ASCII character
d	Displays an integer value
i	Displays an integer value (same as d)
e	Displays a number in scientific notation
f	Displays a floating-point value
g	Displays either scientific notation or floating point, whichever is shorter
o	Displays an octal value
s	Displays a text string

x	Displays a hexadecimal value
X	Displays a hexadecimal value, but using capital letters for A through F

Thus, if you need to display a string variable, you use the format specifier `%s`. If you need to display an integer variable, you use either `%d` or `%i` (`%d` is the C-style for decimals). If you want to display a large value using scientific notation, you use the `%e` format specifier:

```
$ gawk 'BEGIN{
> x = 10 * 100
> printf "The answer is: %e\n", x
> }'
The answer is: 1.000000e+03
$
```

In addition to the control letters, you can use three modifiers for even more control over your output:

- **width:** This is a numeric value that specifies the minimum width of the output field. If the output is shorter, `printf` pads the space with spaces, using right justification for the text. If the output is longer than the specified width, it overrides the width value.
- **prec:** This is a numeric value that specifies the number of digits to the right of the decimal place in floating-point numbers, or the maximum number of characters displayed in a text string.
- **- (minus sign):** The minus sign indicates that left justification should be used instead of right justification when placing data in the formatted space.

When using the `printf` statement, you have complete control over how your output appears. For example, in the “Built-in variables” section, we used the `print` command to display data fields from our records:

```
$ gawk 'BEGIN{FS="\n"; RS="" } {print $1,$4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

You can use the `printf` command to help format the output so it looks better. First, let’s just convert the `print` command to a `printf` command and see what that does:

```
$ gawk 'BEGIN{FS="\n"; RS="" } {printf "%s %s\n", $1, $4}' data2
Riley Mullen (312)555-1234
Frank Williams (317)555-9876
Haley Snell (313)555-4938
$
```

That produces the same output as the `print` command. The `printf` command uses the `%s` format specifier as a placeholder for the two string values.

Notice that you have to manually add the newline character at the end of the `printf` command to force a new line. Without it, the `printf` command uses the same line on subsequent prints.

This is useful if you need to print multiple things on the same line, but using separate `printf` commands:

```
$ gawk 'BEGIN{FS=","} {printf "%s ", $1} END{printf "\n"}' data1
data11 data21 data31
$
```

Both `printf` outputs appear on the same line. To be able to terminate the line, the `END` section prints a single newline character.

Next, let's use a modifier to format the first string value:

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%16s %s\n", $1, $4}' data2
Riley Mullen   (312)555-1234
Frank Williams (317)555-9876
Haley Snell    (313)555-4938
$
```

By adding the 16 modifier value, we force the output for the first string to use 16 spaces. By default, the `printf` command uses right justification to place the data in the format space. To make it left justified, just add a minus sign to the modifier:

```
$ gawk 'BEGIN{FS="\n"; RS=""} {printf "%-16s %s\n", $1, $4}' data2
Riley Mullen      (312)555-1234
Frank Williams    (317)555-9876
Haley Snell       (313)555-4938
$
```

Now that looks pretty professional!

The `printf` command also comes in handy when dealing with floating-point values. By specifying a format for the variable, you can make the output look more uniform:

```
$ gawk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
>     total += $i
> }
> avg = total / 3
> printf "Average: %5.1f\n", avg
> }' data5
```

```
Average: 128.3
Average: 137.7
Average: 176.7
$
```

By using the `%5.1f` format specifier, you can force the `printf` command to round the floating-point values to a single decimal place.

## Built-In Functions

The gawk programming language provides quite a few built-in functions that perform common mathematical, string, and even time functions. You can utilize these functions in your gawk programs to help cut down on the coding requirements in your scripts. This section walks you through the different built-in functions available in gawk.

### Mathematical functions

If you've done programming in any type of language, you're probably familiar with using built-in functions in your code to perform common mathematical functions. The gawk programming language doesn't disappoint those looking for advanced mathematical features.

Table 22-4 shows the mathematical built-in functions available in gawk.

**TABLE 22-4 The gawk Mathematical Functions**

Function	Description
<code>atan2(x, y)</code>	The arctangent of $x / y$ , with $x$ and $y$ specified in radians
<code>cos(x)</code>	The cosine of $x$ , with $x$ specified in radians
<code>exp(x)</code>	The exponential of $x$
<code>int(x)</code>	The integer part of $x$ , truncated toward 0
<code>log(x)</code>	The natural logarithm of $x$
<code>rand()</code>	A random floating point value larger than 0 and less than 1
<code>sin(x)</code>	The sine of $x$ , with $x$ specified in radians
<code>sqrt(x)</code>	The square root of $x$
<code>srand(x)</code>	Specifies a seed value for calculating random numbers

Although it does not have an extensive list of mathematical functions, gawk does provide some of the basic elements you need for standard mathematical processing. The `int()` function produces the integer portion of a value, but it doesn't round the value. It behaves much like a floor function found in other programming languages. It produces the nearest integer to a value between the value and 0.

This means that the `int()` function of the value 5.6 returns 5, while the `int()` function of the value -5.6 returns -5.

The `rand()` function is great for creating random numbers, but you need to use a trick to get meaningful values. The `rand()` function returns a random number, but only between the values 0 and 1 (not including 0 or 1). To get a larger number, you need to scale the returned value.

A common method for producing larger integer random numbers is to create an algorithm that uses the `rand()` function, along with the `int()` function:

```
x = int(10 * rand())
```

This returns a random integer value between (and including) 0 and 9. Just substitute the 10 in the equation with the upper limit value for your application, and you're ready to go.

Be careful when using some of the mathematical functions, because the `gawk` programming language does have a limited range of numeric values it can work with. If you go over that range, you get an error message:

```
$ gawk 'BEGIN{x=exp(100); print x}'
26881171418161356094253400435962903554686976
$ gawk 'BEGIN{x=exp(1000); print x}'
gawk: warning: exp argument 1000 is out of range
inf
$
```

The first example calculates the natural exponential function of 100, which is a very large number but within the range of the system. The second example attempts to calculate the natural exponential function of 1,000, which goes over the numerical range limit of the system and produces an error message.

Besides the standard mathematical functions, `gawk` also provides a few functions for bitwise manipulating of data:

- `and(v1, v2)`: Performs a bitwise AND of values `v1` and `v2`
- `compl(val)`: Performs the bitwise complement of `val`
- `lshift(val, count)`: Shifts the value `val` count number of bits left
- `or(v1, v2)`: Performs a bitwise OR of values `v1` and `v2`
- `rshift(val, count)`: Shifts the value `val` count number of bits right
- `xor(v1, v2)`: Performs a bitwise XOR of values `v1` and `v2`

The bit manipulation functions are useful when working with binary values in your data.



## String functions

The gawk programming language also provides several functions you can use to manipulate string values, shown in Table 22-5.

**TABLE 22-5 The gawk String Functions**

Function	Description
<code>asort(s [,d])</code>	This function sorts an array <i>s</i> based on the data element values. The index values are replaced with sequential numbers indicating the new sort order. Alternatively, the new sorted array is stored in array <i>d</i> if specified.
<code>asorti(s [,d])</code>	This function sorts an array <i>s</i> based on the index values. The resulting array contains the index values as the data element values, with sequential number indexes indicating the sort order. Alternatively, the new sorted array is stored in array <i>d</i> if specified.
<code>gensub(r, s, h [, t])</code>	This function searches either the variable <code>\$0</code> , or the target string <i>t</i> if supplied, for matches of the regular expression <i>r</i> . If <i>h</i> is a string beginning with either <i>g</i> or <i>G</i> , it replaces the matching text with <i>s</i> . If <i>h</i> is a number, it represents which occurrence of <i>r</i> to replace.
<code>gsub(r, s [,t])</code>	This function searches either the variable <code>\$0</code> , or the target string <i>t</i> if supplied, for matches of the regular expression <i>r</i> . If found, it substitutes the string <i>s</i> globally.
<code>index(s, t)</code>	This function returns the index of the string <i>t</i> in string <i>s</i> , or 0 if not found.
<code>length([s])</code>	This function returns the length of string <i>s</i> , or if not specified, the length of <code>\$0</code> .
<code>match(s, r [,a])</code>	This function returns the index of the string <i>s</i> where the regular expression <i>r</i> occurs. If array <i>a</i> is specified, it contains the portion of <i>s</i> that matches the regular expression.
<code>split(s, a [,r])</code>	This function splits <i>s</i> into array <i>a</i> using either the FS character, or the regular expression <i>r</i> if supplied. It returns the number of fields.
<code>sprintf(format, variables)</code>	This function returns a string similar to the output of <code>printf</code> using the <i>format</i> and <i>variables</i> supplied.
<code>sub(r, s [,t])</code>	This function searches either the variable <code>\$0</code> , or the target string <i>t</i> , for matches of the regular expression <i>r</i> . If found, it substitutes the string <i>s</i> for the first occurrence.
<code>substr(s, i [,n])</code>	This function returns the <i>n</i> th character substring of <i>s</i> , starting at index <i>i</i> . If <i>n</i> is not supplied, the rest of <i>s</i> is used.
<code>tolower(s)</code>	This function converts all characters in <i>s</i> to lowercase.
<code>toupper(s)</code>	This function converts all characters in <i>s</i> to uppercase.

Some string functions are relatively self-explanatory:

```
$ gawk 'BEGIN{x = "testing"; print toupper(x); print length(x) }'  
TESTING  
7  
$
```

However, some string functions can get pretty complicated. The `asort` and `asorti` functions are new `gawk` functions that allow you to sort an array variable based on either the data element values (`asort`) or the index values (`asorti`). Here's an example of using `asort`:

```
$ gawk 'BEGIN{  
> var["a"] = 1  
> var["g"] = 2  
> var["m"] = 3  
> var["u"] = 4  
> asort(var, test)  
> for (i in test)  
>   print "Index:",i," - value:",test[i]  
> }'  
Index: 4 - value: 4  
Index: 1 - value: 1  
Index: 2 - value: 2  
Index: 3 - value: 3  
$
```

The new array, `test`, contains the newly sorted data elements of the original array, but the index values are now changed to numerical values, indicating the proper sort order.

The `split` function is a great way to push data fields into an array for further processing:

```
$ gawk 'BEGIN{ FS="," }{  
> split($0, var)  
> print var[1], var[5]  
> }' data1  
data11 data15  
data21 data25  
data31 data35  
$
```

The new array uses sequential numbers for the array index, starting with index value 1 containing the first data field.

## Time functions

The `gawk` programming language contains a few functions to help you deal with time values, shown in Table 22-6.

**TABLE 22-6 The gawk Time Functions**

Function	Description
<code>mktime (datespec)</code>	Converts a date specified in the format YYYY MM DD HH MM SS [DST] into a timestamp value
<code>strftime (format [, timestamp])</code>	Formats either the current time of day timestamp, or timestamp if provided, into a formatted day and date, using the <code>date ()</code> shell function format
<code>systemtime ()</code>	Returns the timestamp for the current time of day

The time functions are often used when working with log files that contain dates that you need to compare. By converting the text representation of a date to the epoch time (the number of seconds since midnight, January 1, 1970), you can easily compare dates.

The following is an example of using the time functions in a gawk program:

```
$ gawk 'BEGIN{
> date = systemtime()
> day = strftime("%A, %B %d, %Y", date)
> print day
> }'
Friday, December 26, 2014
$
```

This example uses the `systemtime` function to retrieve the current epoch timestamp from the system and then uses the `strftime` function to convert it into a human-readable format using the `date` shell command's date format characters.

## User-Defined Functions

You're not limited to just using the built-in functions available in gawk. You can create your own functions for use in gawk programs. This section shows you how to define and use your own functions in gawk programs.

### Defining a function

To define you own function, you must use the `function` keyword:

```
function name([variables])
{
    statements
}
```

The function name must uniquely identify your function. You can pass one or more variables into the function from the calling `gawk` program:

```
function printthird()
{
    print $3
}
```

This function prints the third data field in the record.

The function can also return a value using the `return` statement:

```
return value
```

The value can be a variable, or an equation that evaluates to a value:

```
function myrand(limit)
{
    return int(limit * rand())
}
```

You can assign the value returned from the function to a variable in the `gawk` program:

```
x = myrand(100)
```

The variable contains the value returned from the function.

## Using your functions

When you define a function, it must appear by itself before you define any programming sections (including the `BEGIN` section). This may look a little odd at first, but it helps keep the function code separate from the rest of the `gawk` program:

```
$ gawk '
> function myprint()
> {
>     printf "%-16s - %s\n", $1, $4
> }
> BEGIN{FS="\n"; RS=""}
> {
>     myprint()
> }' data2
Riley Mullen      - (312)555-1234
Frank Williams    - (317)555-9876
Haley Snell       - (313)555-4938
$
```

The function defines the `myprint()` function, which formats the first and fourth data fields in the record for printing. The `gawk` program then uses the function to display the data from the data file.

After you define a function, you can use it as often as necessary in the program section of the code. This saves lots of work when using long algorithms.

## Creating a function library

Obviously, having to rewrite your `gawk` functions every time you need them is not a pleasant experience. However, `gawk` provides a way for you to combine your functions into a single library file that you can use in all your `gawk` programming.

First, you need to create a file that contains all your `gawk` functions:

```
$ cat funclib
function myprint()
{
    printf "%-16s - %s\n", $1, $4
}
function myrand(limit)
{
    return int(limit * rand())
}
function printthird()
{
    print $3
}
$
```

The `funclib` file contains three function definitions. To use them, you need to use the `-f` command line parameter. Unfortunately, you can't combine the `-f` command line parameter with an inline `gawk` script, but you can use multiple `-f` parameters on the same command line.

Thus, to use your library, just create a file that contains your `gawk` program, and specify both the library file and your program file on the command line:

```
$ cat script4
BEGIN{ FS="\n"; RS="" }
{
    myprint()
}
$ gawk -f funclib -f script4 data2
Riley Mullen      - (312)555-1234
Frank Williams    - (317)555-9876
Haley Snell       - (313)555-4938
$
```

Now you just need to add the `funclib` file to your `gawk` command line whenever you need to use a function defined in the library.

## Working through a Practical Example

---

The advanced `gawk` features come in handy if you have to handle data values in a data file, such as tabulating sales figures or calculating bowling scores. When you work with data files, the key is to first group related data records together and then perform any calculations required on the related data.

For example, let's work with a data file that contains the bowling scores from a game between two teams, each with two players:

```
$ cat scores.txt
Rich Blum,team1,100,115,95
Barbara Blum,team1,110,115,100
Christine Bresnahan,team2,120,115,118
Tim Bresnahan,team2,125,112,116
$
```

Each player has scores from three separate games in the data file, and each player is identified by a team name in the second column. Here's the shell script to sort the data for each team and calculate the totals and averages:

```
$ cat bowling.sh
#!/bin/bash

for team in $(gawk -F, '{print $2}' scores.txt | uniq)
do
    gawk -v team=$team 'BEGIN{FS=","; total=0}
    {
        if ($2==team)
        {
            total += $3 + $4 + $5;
        }
    }
    END {
        avg = total / 6;
        print "Total for", team, "is", total, ",the average is",avg
    }' scores.txt
done
$
```

The first `gawk` statement inside the `for` loop filters out the team names in the data file and then uses the `uniq` function to return one value for each separate team name. The `for` loop then iterates for each separate team name.

The gawk statement inside the `for` loop is what's doing the calculations. For each data record, it first determines if the team name matches the loop team. That's done by using the `-v` option in gawk, which allows us to pass a shell variable inside the gawk program. If the team name matches, the code keeps a running sum of the three scores in the data record, adding each data record's values, as long as that data record matches the team name.

At the end of each loop iteration, the gawk code displays the score totals, as well as the average of the scores. The output should look like this:

```
$ ./bowling.sh
Total for team1 is 635, the average is 105.833
Total for team2 is 706, the average is 117.667
$
```

Now you have a handy shell script to calculate the results of all your bowling tournaments; you just need to plug the data from each player into the data text file and run the script!

## Summary

This chapter walked you through the more advanced features of the gawk programming language. Every programming language requires using variables, and gawk is no different. The gawk programming language includes some built-in variables that you can use to reference specific data field values and retrieve information about the number of data fields and records processed in the data file. You can also create your own variables for use in your scripts.

The gawk programming language also provides many of the standard structured commands you expect from a programming language. You can easily create fancy programs using `if-then` logic and `while`, `do-while`, and `for` loops. Each of these commands allows you to alter the flow of your gawk program script to iterate through data field values to create detailed data reports.

The `printf` command is a great tool to have if you need to customize your report output. It allows you to specify the exact format for displaying data from the gawk program script. You can easily create formatted reports, placing data elements in exactly the correct position.

Finally, this chapter discussed the many built-in functions available in the gawk programming language and showed you how to create your own functions. The gawk program contains many useful functions for handling mathematical features, such as standard square roots and logarithms, as well as trigonometric functions. There are also several string-related functions that make extracting substrings from larger strings a breeze.

You aren't limited to the built-in functions in the `gawk` program. If you're working on an application that uses lots of specialized algorithms, you can create your own functions to process the algorithms and use those functions in your own code. You can also set up a library file containing all the functions you use in your `gawk` programs, saving you time and effort in all your coding.

The next chapter switches gears a little. It examines a few other shell environments you may run into in your Linux shell-scripting endeavors. Although the `bash` shell is the most common shell used in Linux, it's not the only shell. It helps to know a little about some of the other shells available and how they differ from the `bash` shell.