Title

stata.com

printf( ) — Format output

Description

`printf()` displays output at the terminal.

`sprintf()` returns a string that can then be displayed at the terminal, written to a file, or used in any other way a string might be used.

Syntax

void `printf(`*string scalar fmt*`,` *r1*`,` *r2*`, ...,` *rN*`)`

string scalar `sprintf(`*string scalar fmt*`,` *r1*`,` *r2*`, ...,` *rN*`)`

where *fmt* may contain a mix of text and %*fmts*, such as

```
printf("The result is %9.2f, adjusted\n", result)
printf("%s = %9.0g\n", name, value)
```

There must be a one-to-one correspondence between the %*fmts* in *fmt* and the number of results to be displayed.

Along with the usual %*fmts* that Stata provides (see [D] **format**), also provided are

Format	Meaning
%f	%11.0f, compressed
%g	%11.0g, compressed
%e	%11.8e, compressed
%s	%#s, # = whatever necessary
%us	%#us, # = whatever necessary
%uds	%#uds, # = whatever necessary

Compressed means that, after the indicated format is applied, all leading and trailing blanks are removed.

C programmers, be warned: %d is Stata's (old) calendar date format (equivalent to modern Stata's %td format) and not an integer format; use %f for formatting integers.

The following character sequences are given a special meaning when contained within *fmt*:

Character sequence	Meaning
%%	one %
\n	newline
\r	carriage return
\t	tab
\\	one \

Remarks and examples

stata.com

Remarks are presented under the following headings:

printf( )

`printf()` displays output at the terminal. A program might contain the line

```
printf("the result is %f\n", result)
```

and display the output

```
the result is 5.213
```

or it might contain the lines

```
printf("{txt}{space 13}{c |}      Coef.    Std. Err.\n")
printf("{hline 13}{c +}{hline 24}\n")
printf("{txt}%12s {c |} {res}%10.0g  %10.0g\n",
  varname[i], coef[i], se[i])
```

and so display the output

```
             |      Coef.    Std. Err.
-------------+------------------------
         mpg |  -.0059541     .0005921
```

Do not forget to include \n at the end of lines. When \n is not included, the line continues. For instance, the code

```
printf("{txt}{space 13}{c |}      Coef.    Std. Err.\n")
printf("{hline 13}{c +}{hline 24}\n")
printf("{txt}%12s {c |} {res}", varname[i])
printf("%10.0g", coef[i])
printf(" ")
printf("%10.0g", se[i])
printf("\n")
```

produces the same output as shown above.

Although users are unaware of it, Stata buffers output. This makes Stata faster. A side effect of the buffering, however, is that output may not appear when you want it to appear. Consider the code fragment

```
for (n=1; !converged(b, b0); n++) {
     printf("iteration %f:  diff = %12.0g\n", n, b-b0)
     b0 = b
      ... new calculation of b ...
}
```

One of the purposes of the iteration output is to keep the user informed that the code is indeed working, yet as the above code is written, the user probably will not see the iteration messages as they occur. Instead, nothing will appear for a while, and then, unexpectedly, many iteration messages will appear as Stata, buffers full, decides to send to the terminal the waiting output.

To force output to be displayed, use [M-5] **displayflush( )**:

```
for (n=1; !converged(b, b0); n++) {
     printf("iteration %f:  diff = %12.0g\n", n, b-b0)
     displayflush()
     b0 = b
      ... new calculation of b ...
}
```

It is only in situations like the above that use of `displayflush()` is necessary. In other cases, it is better to let Stata decide when output buffers should be flushed. (Ado-file programmers: you have never had to worry about this because, at the ado-level, all output is flushed as it is created. Mata, however, is designed to be fast and so `printf()` does not force output to be flushed until it is efficient to do so.)

sprintf( )

The difference between `sprintf()` and `printf()` is that, whereas `printf()` sends the resulting string to the terminal, `sprintf()` returns it. Since Mata displays the results of expressions that are not assigned to variables, `sprintf()` used by itself also displays output:

```
: sprintf("the result is %f\n", result)
  the result is 5.2130a
```

The outcome is a little different from that produced by `printf()` because the output-the-unassigned-expression routine indents results by 2 and displays all the characters in the string (the *0a* at the end is the `\n` newline character). Also, the output-the-unassigned-expression routine does not honor SMCL, electing instead to display the codes:

```
: sprintf("{txt}the result is {res}%f", result)
  {txt}the result is {res}5.213
```

The purpose of `sprintf()` is to create strings that will then be used with `printf()`, with [M-5] **display( )**, with `fput()` (see [M-5] **fopen( )**), or with some other function.

Pretend that we are creating a dynamically formatted table. One of the columns in the table contains integers, and we want to create a %*fmt* that is exactly the width required. That is, if the integers to appear in the table are 2, 9, and 20, we want to create a %2.0f format for the column. We assume the integers are in the column vector dof in what follows:

```
max = 0
for (i=1; i<=rows(dof); i++) {
     len = strlen(sprintf("%f", dof[i])
     if (len>max) max = len
}
fmt = sprintf("%%%f.0f", max)
```

We used sprintf() twice in the above. We first used sprintf() to produce the string representation of the integer dof[i], and we used the %f format so that the length would be whatever was necessary, and no more. We obtained in max the maximum length. If dof contained 2, 9, and 20, by the end of our loop, max will contain 2. We finally used sprintf() to create the %#.0f format that we wanted: %2.0f.

The format string %%%f.0f in the final sprintf() is a little difficult to read. The first two percent signs amount to one real percent sign, so in the output we now have % and we are left with %f.0f. The %f is a format—it is how we are to format max—and so in the output we now have %2, and we are left with .0f. .0f is just a string, so the final output is %2.0f.

The %us and %uds formats

The %*w*us and %*w*uds formats are similar to %*w*s. These formats display a string in a right-justified field of width *w*. %-*w*us and %-*w*uds display the string in a left-justified field. %~*w*us and %~*w*uds display the string center-justified.

The difference between %*w*s, %*w*us, and %*w*uds is how the number of padding spaces is calculated. %*w*s pads the number of spaces to the left of *s* to make the total number of bytes to be *w*. %*w*us pads the number of spaces to the left of *s* to make the total number of Unicode characters to be *w*. %*w*uds pads the number of spaces to the left of *s* to make the total number of display columns to be *w*.

Note that *s* is returned without change if the number of Unicode characters is greater than or equal to *w* in %*w*us or if the number of display columns is greater than or equal to *w* in %*w*uds.

Conformability

printf(*fmt*, r_1, r_2, ..., r_N)

fmt:	1×1
r_1:	1×1
r_2:	1×1
...	
r_N:	1×1
result:	*void*

sprintf(*fmt*, r_1, r_2, ..., r_N)

fmt:	1×1
r_1:	1×1
r_2:	1×1
...	
r_N:	1×1
result:	1×1

Diagnostics

`printf()` and `sprintf()` abort with error if a %*fmt* is misspecified, if a numeric %*fmt* corresponds to a string result or a string %*fmt* to a numeric result, or there are too few or too many %*fmts* in *fmt* relative to the number of *results* specified.

Also see

[M-5] **displayas( )** — Set display level

[M-5] **displayflush( )** — Flush terminal-output buffer

[M-4] **io** — I/O functions