Units Conversion

Edition 1.86 for `units` Version 1.84

Adrian Mariano

1 Overview of `units`

The `units` program converts quantities expressed in various scales to their equivalents in other scales. The `units` program can handle multiplicative scale changes as well as nonlinear conversions such as Fahrenheit to Celsius.[1] Temperature conversions require a special syntax. See [tempconvert], page 6.

The units are defined in an external data file. You can use the extensive data file that comes with this program, or you can provide your own data file to suit your needs.

You can use the program interactively with prompts, or you can use it from the command line.

2 Interacting with `units`

To invoke units for interactive use, type *`units`* at your shell prompt. The program will print something like this:

```
2131 units, 53 prefixes, 24 nonlinear units

You have:
```

At the '`You have:`' prompt, type the quantity and units that you are converting *from*. For example, if you want to convert ten meters to feet, type *`10 meters`*. Next, `units` will print '`You want:`'. You should type the type of units you want to convert *to*. To convert to feet, you would type *`feet`*. Note that if the readline library was compiled in then the tab key can be used to complete unit names. See Chapter 11 [Readline support], page 13, for more information about readline.

The answer will be displayed in two ways. The first line of output, which is marked with a '`*`' to indicate multiplication, gives the result of the conversion you have asked for. The second line of output, which is marked with a '`/`' to indicate division, gives the inverse of the conversion factor. If you convert 10 meters to feet, `units` will print

```
* 32.808399
/ 0.03048
```

which tells you that 10 meters equals about 32.8 feet. The second number gives the conversion in the opposite direction. In this case, it tells you that 1 foot is equal to about 0.03 dekameters since the dekameter is 10 meters. It also tells you that 1/32.8 is about .03.

The `units` program prints the inverse because sometimes it is a more convenient number. In the example above, for example, the inverse value is an exact conversion: a foot is exactly .03048 dekameters. But the number given the other direction is inexact.

If you try to convert grains to pounds, you will see the following:

```
You have: grains
You want: pounds
        * 0.00014285714
        / 7000
```

[1] But Fahrenheit to Celsius is linear, you insist. Not so. A transformation T is linear if $T(x+y) = T(x)+T(y)$ and this fails for $T(x) = ax+b$. This transformation is affine, but not linear.

From the second line of the output you can immediately see that a grain is equal to a seven thousandth of a pound. This is not so obvious from the first line of the output. If you find the output format confusing, try using the '`--verbose`' option:

```
You have: grain
You want: aeginamina
        grain = 0.00010416667 aeginamina
        grain = (1 / 9600) aeginamina
```

If you request a conversion between units which measure reciprocal dimensions, then `units` will display the conversion results with an extra note indicating that reciprocal conversion has been done:

```
You have: 6 ohms
You want: siemens
        reciprocal conversion
        * 0.16666667
        / 6
```

Reciprocal conversion can be suppressed by using the '`--strict`' option. As usual, use the '`--verbose`' option to get more comprehensible output:

```
You have: tex
You want: typp
        reciprocal conversion
        1 / tex = 496.05465 typp
        1 / tex = (1 / 0.0020159069) typp

You have: 20 mph
You want: sec/mile
        reciprocal conversion
        1 / 20 mph = 180 sec/mile
        1 / 20 mph = (1 / 0.0055555556) sec/mile
```

If you enter incompatible unit types, the `units` program will print a message indicating that the units are not conformable and it will display the reduced form for each unit:

```
You have: ergs/hour
You want: fathoms kg^2 / day
conformability error
        2.7777778e-11 kg m^2 / sec^3
        2.1166667e-05 kg^2 m / sec
```

If you only want to find the reduced form or definition of a unit, simply press return at the '`You want:`' prompt. Here is an example:

```
You have: jansky
You want:
        Definition: fluxunit = 1e-26 W/m^2 Hz = 1e-26 kg / s^2
```

The output from `units` indicates that the jansky is defined to be equal to a fluxunit which in turn is defined to be a certain combination of watts, meters, and hertz. The fully reduced (and in this case somewhat more cryptic) form appears on the far right.

Some named units are treated as dimensionless in some situations. These include the radian and steradian. These units will be treated as equal to 1 in units conversions. Power

is equal to torque times angular velocity. This conversion can only be performed if the radian dimensionless.

```
You have: (14 ft lbf) (12 radians/sec)
You want: watts
        * 227.77742
        / 0.0043902509
```

Note that named dimensionaless units are not treated as dimensionless in other contexts. They cannot be used as exponents so for example, '`meter^radian`' is not allowed.

If you want a list of options you can type `?` at the '`You want:`' prompt. The program will display a list of named units which are conformable with the unit that you entered at the '`You have:`' prompt above. Note that conformable unit combinations will not appear on this list.

Typing ***help*** at either prompt displays a short help message. You can also type ***help*** followed by a unit name. This will invoke a pager on the units data base at the point where that unit is defined. You can read the definition and comments that may give more details or historical information about the unit.

3 Using `units` non-interactively

The `units` program can perform units conversions non-interactively from the command line. To do this, type the command, type the original units expression, and type the new units you want. You will probably need to protect the units expressions from interpretation by the shell using single quote characters.

If you type

```
units '2 liters' 'quarts'
```

then `units` will print

```
        * 2.1133764
        / 0.47317647
```

and then exit. The output tells you that 2 liters is about 2.1 quarts, or alternatively that a quart is about 0.47 times 2 liters.

If the conversion is successful, then `units` will return success (0) to the calling environment. If `units` is given non-conformable units to convert, it will print a message giving the reduced form of each unit and it will return failure (nonzero) to the calling environment.

When `units` is invoked with only one argument, it will print out the definition of the specified unit. It will return failure if the unit is not defined and success if the unit is defined.

4 Unit expressions

In order to enter more complicated units or fractions, you will need to use operations such as powers, products and division. Powers of units can be specified using the '`^`' character as shown in the following example, or by simple concatenation: '`cm3`' is equivalent to '`cm^3`'. If the exponent is more than one digit, the '`^`' is required. An exponent like '`2^3^2`' is evaluated right to left. The '`^`' operator has the second highest precedence.

```
You have: cm^3
You want: gallons
        * 0.00026417205
        / 3785.4118

You have: arabicfoot-arabictradepound-force
You want: ft lbf
        * 0.7296
        / 1.370614
```

Multiplication of units can be specified by using spaces, or an asterisk ('`*`'). If `units` is invoked with the '`--product`' option then the hyphen ('`-`') also acts as a multiplication operator. Division of units is indicated by the slash ('`/`') or by '`per`'.

```
You have: furlongs per fortnight
You want: m/s
        * 0.00016630986
        / 6012.8727
```

Multiplication has a higher precedence than division and is evaluated left to right, so '`m/s * s/day`' is equivalent to '`m / s s day`' and has dimensions of length per time cubed. Similarly, '`1/2 meter`' refers to a unit of reciprocal length equivalent to .5/meter, which is probably not what you would intend if you entered that expression. You can indicate division of numbers with the vertical dash ('`|`'). This operator has the highest precedence so the square root of two thirds could be written '`2|3^1|2`'.

```
You have: 1|2 inch
You want: cm
        * 1.27
        / 0.78740157
```

Parentheses can be used for grouping as desired.

```
You have: (1/2) kg / (kg/meter)
You want: league
        * 0.00010356166
        / 9656.0833
```

Prefixes are defined separately from base units. In order to get centimeters, the units database defines '`centi-`' and '`c-`' as prefixes. Prefixes can appear alone with no unit following them. An exponent applies only to the immediately preceding unit and its prefix so that '`cm^3`' or '`centimeter^3`' refer to cubic centimeters but '`centi-meter^3`' refers to hundredths of cubic meters. Only one prefix is permitted per unit, so '`micromicrofarad`' will fail, but '`micro-microfarad`' will work.

For `units`, numbers are just another kind of unit. They can appear as many times as you like and in any order in a unit expression. For example, to find the volume of a box which is 2 ft by 3 ft by 12 ft in steres, you could do the following:

```
You have: 2 ft 3 ft 12 ft
You want: stere
        * 2.038813
        / 0.49048148

You have: $ 5 / yard
You want: cents / inch
        * 13.888889
        / 0.072
```

And the second example shows how the dollar sign in the units conversion can precede the five. Be careful: `units` will interpret '`$5`' with no space as equivalent to dollars^5.

Outside of the SI system, it is often desirable to add values of different units together. You may also wish to use `units` as a calculator that keeps track of units. Sums of conformable units are written with the '`+`' character.

```
You have: 2 hours + 23 minutes + 32 seconds
You want: seconds
        * 8612
        / 0.00011611705

You have: 12 ft + 3 in
You want: cm
        * 373.38
        / 0.0026782366

You have: 2 btu + 450 ft lbf
You want: btu
        * 2.5782804
        / 0.38785542
```

The expressions which are added together must reduce to identical expressions in primitive units, or an error message will be displayed:

```
You have: 12 printerspoint + 4 heredium
                                      ^

Illegal sum of non-conformable units
```

Historically '`-`' has been used for products of units, which complicates its iterpretation in `units`. Because `units` provides several other ways to obtain unit products, and because '`-`' is a subtraction operator in general algebraic expressions, `units` treats the binary '`-`' as a subtraction operator by default. This behavior can be altered using the '`--product`' option which causes `units` to treat the binary '`-`' operator as a product operator. Note that when '`-`' is a multiplication operator it has the same precedence as '`*`', but when '`-`' is a subtraction operator it has the lower precedence as the addition operator.

When '`-`' is used as a unary operator it negates its operand. Regardless of the `units` options, if '`-`' appears after '`(`' or after '`+`' then it will act as a negation operator. So you

can always compute 20 degrees minus 12 minutes by entering '`20 degrees + -12 arcmin`'. You must use this construction when you define new units because you cannot know what options will be in force when your definition is processed.

The '`+`' character sometimes appears in exponents like '`3.43e+8`'. This leads to an ambiguity in an expression like '`3e+2 yC`'. The unit '`e`' is a small unit of charge, so this can be regarded as equivalent to '`(3e+2) yC`' or '`(3 e)+(2 yC)`'. This ambiguity is resolved by always interpreting '`+`' as part of an exponent if possible.

Several built in functions are provided: '`sin`', '`cos`', '`tan`', '`ln`', '`log`', '`log2`', '`exp`', '`acos`', '`atan`' and '`asin`'. The '`sin`', '`cos`', and '`tan`' functions require either a dimensionless argument or an argument with dimensions of angle.

```
You have: sin(30 degrees)
You want:
        Definition: 0.5

You have: sin(pi/2)
You want:
        Definition: 1

You have: sin(3 kg)
                  ^
Unit not dimensionless
```

The other functions on the list require dimensionless arguments. The inverse trigonometric functions return arguments with dimensions of angle.

If you wish to take roots of units, you may use the '`sqrt`' or '`cuberoot`' functions. These functions require that the argument have the appropriate root. Higher roots can be obtained by using fractional exponents:

```
You have: sqrt(acre)
You want: feet
        * 208.71074
        / 0.0047913202

You have: (400 W/m^2 / stefanboltzmann)^(1/4)
You have:
        Definition: 289.80882 K

You have: cuberoot(hectare)
                           ^
Unit not a root
```

Nonlinear units are represented using functional notation. They make possible nonlinear unit conversions such temperature. This is different from the linear units that convert temperature differences. Note the difference below. The absolute temperature conversions are handled by units starting with '`temp`', and you must use functional notation. The temperature differences are done using units starting with '`deg`' and they do not require functional notation.

```
You have: tempF(45)
You want: tempC
        7.2222222

You have: 45 degF
You want: degC
        * 25
        / 0.04
```

Think of '`tempF(x)`' not as a function but as a notation which indicates that '`x`' should have units of '`tempF`' attached to it. See Chapter 8 [Nonlinear units], page 11. The first conversion shows that if it's 45 degrees Fahrehneit outside it's 7.2 degrees Celsius. The second conversions indicates that a change of 45 degrees Fahrenheit corresponds to a change of 25 degrees Celsius.

Some other examples of nonlinears units are ring size and wire gauge. There are numerous different gauges and ring sizes. See the units database for more details. Note that wire gauges with multiple zeroes are signified using negative numbers where two zeroes is -1. Alternatively, you can use the synonyms '`g00`', '`g000`', and so on that are defined in the units database.

```
You have: wiregauge(11)
You want: inches
        * 0.090742002
        / 11.020255

You have: brwiregauge(g00)
You want: inches
        * 0.348
        / 2.8735632

You have: 1 mm
You want: wiregauge
        18.201919
```

5 Invoking `units`

You invoke `units` like this:

```
units [options] [from-unit [to-unit]]
```

If the *from-unit* and *to-unit* are omitted, then the program will use interactive prompts to determine which conversions to perform. See Chapter 2 [Interactive use], page 1. If both *from-unit* and *to-unit* are given, `units` will print the result of that single conversion and then exit. If only *from-unit* appears on the command line, `units` will display the definition of that unit and exit. Units specified on the command line will need to be quoted to protect them from shell interpretation and to group them into two arguments. See Chapter 3 [Command line use], page 3.

The following options allow you to read in an alternative units file, check your units file, or change the output format:

‘`-c`’
‘`--check`’ Check that all units and prefixes defined in the units data file reduce to primitive units. Print a list of all units that cannot be reduced. Also display some other diagnostics about suspicious definitions in the units data file. Note that only definitions active in the current locale are checked.

‘`--check-verbose`’
Like the ‘`-check`’ option, this option prints a list of units that cannot be reduced. But to help find unit definitions that cause endless loops, it lists the units as they are checked. If `units` hangs, then the last unit to be printed has a bad definition. Note that only definitions active in the current locale are checked.

‘`-o format`’
‘`--output-format format`’
Use the specified format for numeric output. Format is the same as that for the printf function in the ANSI C standard. For example, if you want more precision you might use ‘`-o %.15g`’.

‘`-f filename`’
‘`--file filename`’
Instruct `units` to load the units file `filename`. If `filename` is the empty string (‘`-f ''`’) then the default units file will be loaded. This enables you to load the default file plus a personal units file. Up to 25 units files may be specified on the command line. This option overrides the `UNITSFILE` environment variable.

‘`-h`’
‘`--help`’ Print out a summary of the options for `units`.

‘`-q`’
‘`--quiet`’
‘`--silent`’
Suppress prompting of the user for units and the display of statistics about the number of units loaded.

‘`-s`’
‘`--strict`’
Suppress conversion of units to their reciprocal units. For example, `units` will normally convert hertz to seconds because these units are reciprocals of each other. The strict option requires that units be strictly conformable to perform a conversion, and will give an error if you attempt to convert hertz to seconds.

‘`-t`’
‘`--terse`’ Give terse output when converting units. This option can be used when calling `units` from another program so that the output is easy to parse.

‘`-v`’
‘`--verbose`’
Give slightly more verbose output when converting units. When combined with the ‘`-c`’ option this gives the same effect as ‘`--check-verbose`’.

‘-V’
‘--version’
: Print program version number, tell whether the readline library has been included, and give the location of the default units data file.

6 Unit definitions

The conversion information is read from a units data file which is called ‘units.dat’ and is probably located in the ‘/usr/local/share’ directory. If you invoke units with the ‘-V’ option, it will print the location of this file. The default file includes definitions for all familiar units, abbreviations and metric prefixes. It also includes many obscure or archaic units.

Many constants of nature are defined, including these:

pi	ratio of circumference to diameter
c	speed of light
e	charge on an electron
force	acceleration of gravity
mole	Avogadro’s number
water	pressure per unit height of water
Hg	pressure per unit height of mercury
au	astronomical unit
k	Boltzman’s constant
mu0	permeability of vacuum
epsilon0	permitivity of vacuum
G	Gravitational constant
mach	speed of sound

The database includes atomic masses for all of the elements and numerous other constants. Also included are the densities of various ingredients used in baking so that ‘2 cups flour_sifted’ can be converted to ‘grams’. This is not an exhaustive list. Consult the units data file to see the complete list, or to see the definitions that are used.

The unit ‘pound’ is a unit of mass. To get force, multiply by the force conversion unit ‘force’ or use the shorthand ‘lbf’. (Note that ‘g’ is already taken as the standard abbreviation for the gram.) The unit ‘ounce’ is also a unit of mass. The fluid ounce is ‘fluidounce’ or ‘floz’. British capacity units that differ from their US counterparts, such as the British Imperial gallon, are prefixed with ‘br’. Currency is prefixed with its country name: ‘belgiumfranc’, ‘britainpound’.

The US Survey foot, yard, and mile can be obtained by using the ‘US’ prefix. These units differ slightly from the international length units. They were in general use until 1959, and are still used for geographic surveys. The acre is officially defined in terms of the US Survey foot. If you want an acre defined according to the international foot, use ‘intacre’. The difference between these units is about 4 parts per million. The British also used a slightly different length measure before 1959. These can be obtained with the prefix ‘UK’.

When searching for a unit, if the specified string does not appear exactly as a unit name, then the units program will try to remove a trailing ‘s’ or a trailing ‘es’. If that fails, units will check for a prefix. All of the standard metric prefixes are defined.

To find out what units and prefixes are available, read the standard units data file.

7 Defining new units

All of the units and prefixes that `units` can convert are defined in the units data file. If you want to add your own units, you can supply your own file.

A unit is specified on a single line by giving its name and an equivalence. Comments start with a '`#`' character, which can appear anywhere in a line. The backslash character ('\') acts as a continuation character if it appears as the last character on a line, making it possible to spread definitions out over several lines if desired. A file can be included by giving the command '`!include`' followed by the file's name. The file will be sought in the same directory as the parent file unless a full path is given.

Unit names must not contain any of the operator characters '`+`', '`-`', '`*`', '`/`', '`|`', '`^`' or the parentheses. They cannot begin with a digit or a decimal point ('`.`'), nor can they end with a digit (except for zero). Be careful to define new units in terms of old ones so that a reduction leads to the primitive units, which are marked with '`!`' characters. Dimensionless units are indicated by using the string '`!dimensionless`' for the unit definition.

When adding new units, be sure to use the '`-c`' option to check that the new units reduce properly. If you create a loop in the units definitions, then `units` will hang when invoked with the '`-c`' options. You will need to use the '`--check-verbose`' option which prints out each unit as it checks them. The program will still hang, but the last unit printed will be the unit which caused the infinite loop.

If you define any units which contain '`+`' characters, carefully check them because the '`-c`' option will not catch non-conformable sums. Be careful with the '`-`' operator as well. When used as a binary operator, the '`-`' character can perform addition or multiplication depending on the options used to invoke `units`. To ensure consistent behavior use '`-`' only as a unary negation operator when writing units definitions. To multiply two units leave a space or use the '`*`' operator. To compute the difference of '`foo`' and '`bar`' write '`foo+(-bar)`' or even '`foo+-bar`'.

Here is an example of a short units file that defines some basic units:

```
m       !               # The meter is a primitive unit
sec     !               # The second is a primitive unit
rad     !dimensionless  # A dimensionless primitive unit
micro-  1e-6            # Define a prefix
minute  60 sec          # A minute is 60 seconds
hour    60 min          # An hour is 60 minutes
inch    0.0254 m        # Inch defined in terms of meters
ft      12 inches       # The foot defined in terms of inches
mile    5280 ft         # And the mile
```

A unit which ends with a '`-`' character is a prefix. If a prefix definition contains any '`/`' characters, be sure they are protected by parentheses. If you define '`half- 1/2`' then '`halfmeter`' would be equivalent to '`1 / 2 meter`'.

8 Defining nonlinear units

Some units conversions of interest are nonlinear; for example, temperature conversions between the Fahrenheit and Celsius scales cannot be done by simply multiplying by conversions factors.

When you give a linear unit definition such as '`inch 2.54 cm`' you are providing information that `units` uses to convert values in inches into primitive units of meters. For nonlinear units, you give a functional definition that provides the same information.

Nonlinear units are represented using a functional notation. It is best to regard this notation not as a function call but as a way of adding units to a number, much the same way that writing a linear unit name after a number adds units to that number. Internally, nonlinear units are defined by a pair of functions which convert to and from linear units in the data file, so that an eventual conversion to primitive units is possible.

Here is an example nonlinear unit definition:

```
tempF(x) [1;K] (x+(-32)) degF + stdtemp ; (tempF+(-stdtemp))/degF + 32
```

A nonlinear unit definition comprises a unit name, a dummy parameter name, two functions, and two corresponding units. The functions tell `units` how to convert to and from the new unit. In order to produce valid results, the arguments of these functions need to have the correct dimensions. To facilitate error checking, you may specify the dimensions.

The definition begins with the unit name followed immediately (with no spaces) by a '`(`' character. In parentheses is the name of the parameter. Next is an optional specification of the units required by the functions in this definition. In the example above, the '`tempF`' function requires an input argument conformable with '`1`'. For normal nonlinear units definitions the forward function will always take a dimensionless argument. The inverse function requires an input argument conformable with '`K`'. In general the inverse function will need units that match the quantity measured by your nonlinear unit. The sole purpose of the expression in brackets to enable `units` to perform error checking on function arguments.

Next the function definitions appear. In the example above, the '`tempF`' function is defined by

```
tempF(x) = (x+(-32)) degF + stdtemp
```

This gives a rule for converting '`x`' in the units '`tempF`' to linear units of absolute temperature, which makes it possible to convert from tempF to other units.

In order to make conversions to Fahrenheit possible, you must give a rule for the inverse conversions. The inverse will be '`x(tempF)`' and its definition appears after a '`;`' character. In our example, the inverse is

```
x(tempF) = (tempF+(-stdtemp))/degF + 32
```

This inverse definition takes an absolute temperature as its argument and converts it to the Fahrenheit temperature. The inverse can be omitted by leaving out the '`;`' character, but then conversions to the unit will be impossible. If the inverse is omitted then the '`--check`' option will display a warning. It is up to you to calculate and enter the correct inverse function to obtain proper conversions. The '`--check`' option tests the inverse at one point and print an error if it is not valid there, but this is not a guarantee that your inverse is correct.

If you wish to make synonyms for nonlinear units, you still need to define both the forward and inverse functions. Inverse functions can be obtained using the '~' operator. So to create a synonym for 'tempF' you could write

```
fahrenheit(x) [1;K] tempF(x); ~tempF(fahrenheit)
```

You may occasionally wish to define a function that operates on units. This can be done using a nonlinear unit definition. For example, the definition below provides conversion between radius and the area of a circle. Note that this definition requires a length as input and produces an area as output, as indicated by the specification in brackets.

```
circlearea(r) [m;m^2] pi r^2 ; sqrt(circlearea/pi)
```

Sometimes you may be interested in a piecewise linear unit such as many wire gauges. Piecewise linear units can be defined by specifying conversions to linear units on a list of points. Conversion at other points will be done by linear interpolation. A partial definition of zinc gauge is

```
zincgauge[in] 1 0.002, 10 0.02, 15 0.04, 19 0.06, 23 0.1
```

In this example, 'zincgauge' is the name of the piecewise linear unit. The definition of such a unit is indicated by the embedded '[' character. After the bracket, you should indicate the units to be attached to the numbers in the table. No spaces can appear before the ']' character, so a definition like 'foo[kg meters]' is illegal; instead write 'foo[kg*meters]'. The definition of the unit consists of a list of pairs optionally separated by commas. This list defines a function for converting from the piecewise linear unit to linear units. The first item in each pair is the function argument; the second item is the value of the function at that argument (in the units specified in brackets). In this example, we define 'zincgauge' at five points. For example, we set 'zincgauge(1)' equal to '0.002 in'. Definitions like this may be more readable if written using continuation characters as

```
zincgauge[in] \
     1 0.002  \
    10 0.02   \
    15 0.04   \
    19 0.06   \
    23 0.1
```

With the preceeding definition, the following conversion can be performed:

```
You have: zincgauge(10)
You want: in
    * 0.02
    / 50
You have: .01 inch
You want: zincgauge
    5
```

If you define a piecewise linear unit that is not strictly monotonic, then the inverse will not be well defined. If the inverse is requested for such a unit, `units` will return the smallest inverse. The '`--check`' option will print a warning if a non-monotonic piecewise linear unit is encountered.

9 Localization

Some units have different values in different locations. The localization feature accomodates this by allowing the units database to specify region dependent definitions. A locale region in the units database begins with '`!locale`' followed by the name of the locale. The leading '`!`' must appear in the first column of the units database. The locale region is terminated by '`!endlocale`'. The following example shows how to define a couple units in a locale.

```
!locale en_GB
ton                     brton
gallon                  brgallon
!endlocale
```

The current locale is specified by the `LOCALE` environment variable. Note that the '`-c`' option only checks the definitions which are active for the current locale.

10 Environment variables

The `units` programs uses the following environment variables.

'`LOCALE`' Specifies the locale. The default is '`en_US`'. Sections of the units database are specific to certain locales.

'`PAGER`' Specifies the pager to use for help and for displaying the conformable units. The help function browses the units database and calls the pager using the `+nn` syntax for specifying a line number. The default pager is `more`, but `less`, `emacs`, or `vi` are possible alternatives.

'`UNITSFILE`'
Specifies the units database file to use (instead of the default). This will be overridden by the '`-f`' option. Note that you can only specify a single units database using this environment variable.

11 Readline support

If the `readline` package has been compiled in, then when `units` is used interactively, numerous command line editing features are available. To check if your version of `units` includes the readline, invoke the program with the '`--version`' option.

For complete information about readline, consult the documentation for the readline package. Without any configuration, `units` will allow editing in the style of emacs. Of particular use with `units` are the completion commands.

If you type a few characters and then hit '`ESC`' followed by the `?` key then `units` will display a list of all the units which start with the characters typed. For example, if you type *`metr`* and then request completion, you will see something like this:

```
You have: metr
metre             metriccup         metrichorsepower  metrictenth
metretes          metricfifth       metricounce       metricton
metriccarat       metricgrain       metricquart       metricyarncount
You have: metr
```

If there is a unique way to complete a unitname, you can hit the tab key and `units` will provide the rest of the unit name. If `units` beeps, it means that there is no unique completion. Pressing the tab key a second time will print the list of all completions.

Index

Table of Contents