Units Conversion

Edition 2.25 for `units` Version 2.25

Adrian Mariano

This manual is for GNU Units (version 2.25), which performs units conversions and units calculations.

Table of Contents

1 Overview of units

The units program converts quantities expressed in various systems of measurement to their equivalents in other systems of measurement. Like many similar programs, it can handle multiplicative scale changes. It can also handle nonlinear conversions such as Fahrenheit to Celsius;[1] see Section 6.1 [Temperature Conversions], page 19. The program can also perform conversions from and to sums of units, such as converting between meters and feet plus inches.

Basic operation is simple: you enter the units that you want to convert *from* and the units that you want to convert *to*. You can use the program interactively with prompts, or you can use it from the command line.

Beyond simple unit conversions, units can be used as a general-purpose scientific calculator that keeps track of units in its calculations. You can form arbitrary complex mathematical expressions of dimensions including sums, products, quotients, powers, and even roots of dimensions. Thus you can ensure accuracy and dimensional consistency when working with long expressions that involve many different units that may combine in complex ways; for an illustration, see Section 5.6 [Complicated Unit Expressions], page 15.

The units are defined in several external data files. You can use the extensive data files that come with the program, or you can provide your own data file to suit your needs. You can also use your own data file to supplement the standard data files.

You can change the default behavior of units with various options given on the command line. See Chapter 10 [Invoking Units], page 35, for a description of the available options.

2 Interacting with units

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

```
Currency exchange rates from FloatRates (USD base) on 2023-07-08
3612 units, 109 prefixes, 122 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 units you want to convert *to*. To convert to feet, you would type *feet*. If the readline library was compiled in, then tab will complete unit names. See Chapter 20 [Readline Support], page 61, for more information about readline. To quit the program type *quit* or *exit* at either prompt.

The result 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

[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—see https://en.wikipedia.org/wiki/Linear_map.

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 0.03.

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 0.03048 dekameters. But the number given the other direction is inexact.

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

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

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 that 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 `Enter` 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. If the ultimate definition and the fully reduced form are identical, the latter is not shown:

```
You have: B
You want:
        Definition: byte = 8 bit
```

The fully reduced form *is* shown if it and the ultimate definition are equivalent but not identical:

```
You have: N
You want:
        Definition: newton = kg m / s^2 = 1 kg m / s^2
```

Some named units are treated as dimensionless in some situations. These units 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 is dimensionless.

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

It is also possible to compute roots and other non-integer powers of dimensionless units; this allows computations such as the altitude of geosynchronous orbit:

```
You have: cuberoot(G earthmass / (circle/siderealday)^2) - earthradius
You want: miles
        * 22243.267
        / 4.4957425e-05
```

Named dimensionless units are not treated as dimensionless in other contexts. They cannot be used as exponents so for example, '`meter^radian`' is forbidden.

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

2.1 Interactive Commands

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 database 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. If your pager allows, you may want to scroll backwards, e.g. with 'b', because sometimes a longer comment about a unit or group of units will appear before the definition. You can generally quit out of the pager by pressing 'q'.

Typing *search text* will display a list of all units whose names contain *text* as a substring along with their definitions. This may help in the case where you aren't sure of the right unit name. Typing *search .* will display a list of all units; this can be helpful if you aren't sure of an appropriate substring, though examining the output can be tedious. Here '.' is a special argument to the *search* command, not a regular expression. If you want to use a regular expression to search units *and* their definitions, the `--list-units` option will display a list of all units, which can be sent to a filter such as `grep`. Typing *search _* will display a list of all assigned runtime variables—whose names begin with '_'—without including all units whose names simply *contain* '_'. See Section 5.7 [Variables Assigned at Run Time], page 16, for more information on runtime variables.

Typing *version* will display the `units` version number; unlike the `--version` option, only the version number is shown, regardless of any other settings. Full information is available with the *info* command.

Typing *info* will display the same information as the `--info` option, described in Chapter 10 [Invoking Units], page 35.

Many values set by command-line options can also be set interactively by typing *set option=value*; typing *set option* will show the value for that option. Typing *set* will show a list of options that can be set; options set to other than default values will have a prepended '*'. See Chapter 11 [Setting Options Interactively], page 41, for more information.

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 unit expression, and type the new units you want. If a units expression contains non-alphanumeric characters, you may need to protect it from interpretation by the shell using single or double 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.

`units` does not require a space between a numerical value and the unit, so the previous example can be given as

```
units 2liters quarts
```

to avoid having to quote the first argument.

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

If the `--conformable` option is given, only one unit expression is allowed, and `units` will print all units conformable with that expression; it is equivalent to giving *?* at the '`You want:`' prompt. For example,

```
units --conformable gauss
B_FIELD   tesla
Gs        gauss
T         tesla
gauss     abvolt sec / cm^2
stT       stattesla
statT     stattesla
stattesla statWb/cm^2
tesla     Wb/m^2
```

If you give more than one unit expression with the `--conformable` option, the program will exit with an error message and return failure. This option has no effect in interactive mode.

If the `--terse` (`-t`) option is given with the `--conformable` option, conformable units are shown without definitions; with the previous example, this would give

```
units --terse --conformable gauss
B_FIELD
Gs
T
gauss
stT
statT
stattesla
tesla
```

When the `--conformable` option is not given and you invoke `units` with only one argument, `units` will print the definition of the specified unit. It will return failure if the unit is not defined and success if the unit is defined.

3.1 Finding Units

The `units` database includes many units, and it can sometimes be difficult to remember every name. The interactive *search* command examines only unit names, so it cannot be used to find units whose *definitions* contain a pattern. If `units` is invoked with the `--list-units` option, the output can be sent to a filter such as `grep`, examining the unit name *and* the definition. For example,

```
units --list-units | grep ';'
dms                                   deg;arcmin;arcsec
ftin                                  ft;in;1|8 in
ftin16                                ft;in;1|16 in
ftin32                                ft;in;1|32 in
ftin4                                 ft;in;1|4 in
ftin64                                ft;in;1|64 in
ftin8                                 ft;in;1|8 in
hms                                   hr;min;sec
inchfine                              in;1|8 in;1|16 in;1|32 in;1|64 in
time                                  year;day;hr;min;sec
uswt                                  lb;oz
  . . .
```

could find all unit lists, which include at least one semicolon in their definitions.

The *search* command doesn't support regular expressions, but with the `--list-units` option the output can be sent to a filter such as `grep`, which does. If you expect a long list of results, you can send the output of the filter to a pager. For example,

```
units --list-units | grep 'paper[[:space:]]' | more
A0paper                               841 mm 1189 mm
A10paper                              26 mm 37 mm
A1paper                               594 mm 841 mm
A2paper                               420 mm 594 mm
A3paper                               297 mm 420 mm
A4paper                               210 mm 297 mm
A5paper                               148 mm 210 mm
A6paper                               105 mm 148 mm
A7paper                               74 mm 105 mm
A8paper                               52 mm 74 mm
A9paper                               37 mm 52 mm
  . . .
```

could show a list of all defined paper sizes.

4 Unit Definitions

The conversion information is read from several units data files: `definitions.units`, `elements.units`, `currency.units`, and `cpi.units`, which are usually located in the `/usr/share/units` directory. If you invoke `units` with the `-V` option, it will print the location of these files. The default main file includes definitions for all familiar units, abbreviations and metric prefixes. It also includes many obscure or archaic units. Many common spelled-out numbers (e.g., '`seventeen`') are recognized.

4.1 Physical Constants

Many constants of nature are defined, including these:

`pi` ratio of circumference of a circle to its 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	permittivity of vacuum
G	Gravitational constant
mach	speed of sound

The standard data file includes 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.

4.2 Atomic Masses of the Elements

The data file `elements.units` includes atomic masses for most elements and most known isotopes. If the mole fractions of constituent isotopes are known, an elemental mass is calculated from the sum of the products of the mole fractions and the masses of the constituent isotopes. If the mole fractions are not known, the mass of the most stable isotope—if known—is given as the elemental mass. For radioactive elements with atomic numbers 95 or greater, the mass number of the most stable isotope is not specified, because the list of studied isotopes is still incomplete. If no stable isotope is known, no elemental mass is given, and you will need to choose the most appropriate isotope.

The data are obtained from the US National Institute for Standards and Technology (NIST): `https://physics.nist.gov/cgi-bin/Compositions/stand_alone.pl?ele=&all=all&ascii=ascii2&isotype=all`. The `elements.units` file can be generated from these data using the `elemcvt` command included with the distribution.

4.3 Currency Exchange Rates and Consumer Price Index

The data file `currency.units` includes currency conversion rates; the file `cpi.units` includes the US Consumer Price Index (CPI), published by the US Bureau of Labor Statistics. The data are updated monthly by the BLS; see Chapter 21 [Updating Currency Exchange Rates and CPI], page 62, for information on updating `currency.units` and `cpi.units`.

4.4 English Customary Units

English customary units differ in various ways among different regions. In Britain a complex system of volume measurements featured different gallons for different materials such as a wine gallon and ale gallon that different by twenty percent. This complexity was swept away in 1824 by a reform that created an entirely new gallon, the British Imperial gallon defined as the volume occupied by ten pounds of water. Meanwhile in the USA the gallon is derived from the 1707 Winchester wine gallon, which is 231 cubic inches. These gallons differ by about twenty percent. By default if `units` runs in the '`en_GB`' locale you will get the British volume measures. If it runs in the '`en_US`' locale you will get the US volume measures. In other locales the default values are the US definitions. If you wish to force different definitions, then set the environment variable `UNITS_ENGLISH` to either '`US`' or '`GB`' to set the desired definitions independent of the locale.

Before 1959, the value of a yard (and other units of measure defined in terms of it) differed slightly among English-speaking countries. In 1959, Australia, Canada, New Zealand, the United Kingdom, the United States, and South Africa adopted the Canadian value of 1 yard = 0.9144 m (exactly), which was approximately halfway between the values used by the UK and the US; it had the additional advantage of making 1 inch = 2.54 cm (exactly). This new standard was termed the *International Yard*. Australia, Canada, and the UK then defined all customary lengths in terms of the International Yard (Australia did not define the furlong or rod); because many US land surveys were in terms of the pre-1959 units, the US continued to define customary surveyors' units (furlong, chain, rod, pole, perch, and link) in terms of the previous value for the foot, which was termed the *US survey foot*. The US defined a *US survey mile* as 5280 US survey feet, and defined a *statute mile* as a US survey mile. The US values for these units differed from the international values by about 2 ppm.

The 1959 redefinition of the foot was legally binding in the US but allowed continued use of the previous definition of the foot for geodetic surveying. It was assumed that this use would be temporary, but use persisted, leading to confusion and errors, and it was at odds with the intent of uniform standards. Since January 1, 2023, the US survey foot has been officially deprecated (85 FR 62698), with its use limited to historical and legacy applications.

The `units` program has always used the international values for these units; the legacy US values can be obtained by using either the '`US`' or the '`survey`' prefix. In either case, the simple familiar relationships among the units are maintained, e.g., 1 '`furlong`' = 660 '`ft`', and 1 '`USfurlong`' = 660 '`USft`', though the metric equivalents differ slightly between the two cases. The '`US`' prefix or the '`survey`' prefix can also be used to obtain the US survey mile and the value of the US yard prior to 1959, e.g., '`USmile`' or '`surveymile`' (but *not* '`USsurveymile`'). To get the US value of the statute mile, use either '`USstatutemile`' or '`USmile`'. The pre-1959 UK values for these units can be obtained with the prefix '`UK`'.

Except for distances that extend over hundreds of miles (such as in the US State Plane Coordinate System), the differences in the miles are usually insignificant:

```
You have: 100 surveymile - 100 mile
You want: inch
        * 12.672025
        / 0.078913984
```

The US acre was officially defined in terms of the US survey foot, but `units` has used a definition based on the international foot; the `units` definition is now the same as the official US value. If you want the previous US acre, use '`USacre`' and similarly use '`USacrefoot`' for the previous US version of that unit. The difference between these units is about 4 parts per million.

4.5 Miscellaneous Notes on Unit Definitions

The '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'. When British capacity units differ from their US counterparts, such as the British Imperial gallon, the unit is defined both ways with 'br' and 'us' prefixes. Your locale settings will determine the value of the unprefixed unit. Currency is prefixed with its country name: 'belgiumfranc', 'britainpound'.

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', 'es'. Next units will replace a trailing 'ies' with 'y'. If that fails, units will check for a prefix. The database includes all of the standard metric prefixes. Only one prefix is permitted per unit, so 'micromicrofarad' will fail. However, prefixes can appear alone with no unit following them, so 'micro*microfarad' will work, as will 'micro microfarad'.

To find out which units and prefixes are available, read the default units data files; the main data file is extensively annotated.

5 Unit Expressions

5.1 Operators

You can enter more complicated units by combining units with operations such as multiplication, division, powers, addition, subtraction, and parentheses for grouping. You can use the customary symbols for these operators when units is invoked with its default options. Additionally, units supports some extensions, including high priority multiplication using a space, and a high priority numerical division operator ('|') that can simplify some expressions.

You multiply units using a space or an asterisk ('*'). The next example shows both forms:

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

You can divide units using the slash ('/') or with 'per':

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

You can use parentheses for grouping:

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

White space surrounding operators is optional, so the previous example could have used '`(1/2)kg/(kg/meter)`'. As a consequence, however, hyphenated spelled-out numbers (e.g., '`forty-two`') cannot be used; '`forty-two`' is interpreted as '`40 - 2`'.

Multiplication using a space has a higher precedence than division using a slash and is evaluated left to right; in effect, the first '`/`' character marks the beginning of the denominator of a unit expression. This makes it simple to enter a quotient with several terms in the denominator: '`J / mol K`'. The '`*`' and '`/`' operators have the same precedence, and are evaluated left to right; if you multiply with '`*`', you must group the terms in the denominator with parentheses: '`J / (mol * K)`'.

The higher precedence of the space operator may not always be advantageous. For example, '`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 0.5/meter, perhaps not what you would intend if you entered that expression. The get a half meter you would need to use parentheses: '`(1/2) meter`'. The '`*`' operator is convenient for multiplying a sequence of quotients. For example, '`m/s * s/day`' is equivalent to '`m/day`'. Similarly, you could write '`1/2 * meter`' to get half a meter.

The `units` program supports another option for numerical fractions: you can indicate division of *numbers* with the vertical bar ('`|`'), so if you wanted half a meter you could write '`1|2 meter`'. You cannot use the vertical bar to indicate division of non-numerical units (e.g., '`m|s`' results in an error message).

Powers of units can be specified using the '`^`' character, as shown in the following example, or by simple concatenation of a unit and its exponent: '`cm3`' is equivalent to '`cm^3`'; if the exponent is more than one digit, the '`^`' is required. You can also use '`**`' as an exponent operator.

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

Concatenation only works with a single unit name: if you write '`(m/s)2`', `units` will treat it as multiplication by 2. When a unit includes a prefix, exponent operators apply to the combination, so '`centimeter3`' gives cubic centimeters. If you separate the prefix from the unit with any multiplication operator (e.g., '`centi meter^3`'), the prefix is treated as a separate unit, so the exponent applies only to the unit without the prefix. The second example is equivalent to '`centi * (meter^3)`', and gives a hundredth of a cubic meter, not a cubic centimeter. The `units` program is limited internally to products of 99 units; accordingly, expressions like '`meter^100`' or '`joule^34`' (represented internally as '`kg^34 m^68 / s^68`') will fail.

The '`|`' operator has the highest precedence, so you can write the square root of two thirds as '`2|3^1|2`'. The '`^`' operator has the second highest precedence, and is evaluated right to left, as usual:

```
You have: 5 * 2^3^2
You want:
        Definition: 2560
```

With a dimensionless base unit, any dimensionless exponent is meaningful (e.g., '`pi^exp(2.371)`'). Even though angle is sometimes treated as dimensionless, exponents cannot have dimensions of angle:

```
You have: 2^radian
                 ^

Exponent not dimensionless
```

If the base unit is not dimensionless, the exponent must be a rational number p/q, and the dimension of the unit must be a power of q, so '`gallon^2|3`' works but '`acre^2|3`' fails. An exponent using the slash ('`/`') operator (e.g., '`gallon^(2/3)`') is also acceptable; the parentheses are needed because the precedence of '`^`' is higher than that of '`/`'. Since `units` cannot represent dimensions with exponents greater than 99, a fully reduced exponent must have $q < 100$. When raising a non-dimensionless unit to a power, `units` attempts to convert a decimal exponent to a rational number with $q < 100$. If this is not possible `units` displays an error message:

```
You have: ft^1.234
Base unit not dimensionless; rational exponent required
```

A decimal exponent must match its rational representation to machine precision, so '`acre^1.5`' works but '`gallon^0.666`' does not.

5.2 Sums and Differences of Units

You may sometimes want to add values of different units that are outside the SI. You may also wish to use `units` as a calculator that keeps track of units. Sums of conformable units are written with the '`+`' character, and differences 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 that are added or subtracted must reduce to identical expressions in primitive units, or an error message will be displayed:

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

Invalid sum of non-conformable units
```

If you add two values of vastly different scale you may exceed the available precision of floating point (about 15 digits). The effect is that the addition of the smaller value makes no change to the larger value; in other words, the smaller value is treated as if it were zero.

```
You have: lightyear + cm
```

No warning is given, however. As usual, the precedence for '+' and '-' is lower than that of the other operators. A fractional quantity such as 2 1/2 cups can be given as '(2+1|2) cups'; the parentheses are necessary because multiplication has higher precedence than addition. If you omit the parentheses, `units` attempts to add '2' and '1|2 cups', and you get an error message:

```
You have: 2+1|2 cups
                   ^
Invalid sum or difference of non-conformable units
```

The expression could also be correctly written as '(2+1/2) cups'. If you write '2 1|2 cups' the space is interpreted as *multiplication* so the result is the same as '1 cup'.

The '+' and '-' characters 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 '+' and '-' as part of an exponent if possible.

5.3 Numbers as Units

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 that 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 'dollar^5'.

5.4 Built-in Functions

Several built-in functions are provided: 'sin', 'cos', 'tan', 'secant', 'csc', 'cot', 'asin', 'acos', 'atan', 'asecant', 'acsc', 'acot', 'sinh', 'cosh', 'tanh', 'sech', 'csch', 'coth', 'asinh', 'acosh', 'atanh', 'asech', 'acsch', 'acoth', 'exp', 'ln', 'log', 'abs', 'round', 'floor', 'ceil', 'factorial', 'Gamma', 'lnGamma', 'erf', and 'erfc'; the function 'lnGamma' is the natural logarithm of the 'Gamma' function.

The 'sin', 'cos', 'tan', 'secant', 'csc' and 'cot' 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.

The '`ln`' and '`log`' functions give natural log and log base 10 respectively. To obtain logs for any integer base, enter the desired base immediately after '`log`'. For example, to get log base 2 you would write '`log2`' and to get log base 47 you could write '`log47`'.

```
You have: log2(32)
You want:
        Definition: 5
You have: log3(32)
You want:
        Definition: 3.1546488
You have: log4(32)
You want:
        Definition: 2.5
You have: log32(32)
You want:
        Definition: 1
You have: log(32)
You want:
        Definition: 1.50515
You have: log10(32)
You want:
        Definition: 1.50515
```

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. You can obtain higher roots 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
```

5.5 Previous Result

You can insert the result of the previous conversion using the underscore ('`_`'). It is useful when you want to convert the same input to several different units, for example

```
You have: 2.3 tonrefrigeration
You want: btu/hr
        * 27600
        / 3.6231884e-005
You have: _
You want: kW
        * 8.0887615
        / 0.12362832
```

Suppose you want to do some deep frying that requires an oil depth of 2 inches. You have 1/2 gallon of oil, and want to know the largest-diameter pan that will maintain the required depth. The nonlinear unit '`circlearea`' gives the *radius* of the circle (see Section 6.3 [Other Nonlinear Units], page 21, for a more detailed description) in SI units; you want the *diameter* in *inches*:

```
You have: 1|2 gallon / 2 in
You want: circlearea
        0.10890173 m
You have: 2 _
You want: in
        * 8.5749393
        / 0.1166189
```

In most cases, surrounding white space is optional, so the previous example could have used '`2_`'. If '`_`' follows a non-numerical unit symbol, however, the space is required:

```
You have: m_
           ^

Parse error
```

You can use the '`_`' symbol any number of times; for example,

```
You have: m
You want:
        Definition: 1 m
You have: _ _
You want:
        Definition: 1 m^2
```

Using '_' before a conversion has been performed (e.g., immediately after invocation) generates an error:

```
You have: _
          ^

No previous result; '_' not set
```

Accordingly, '_' serves no purpose when `units` is invoked non-interactively.

If `units` is invoked with the `--verbose` option (see Chapter 10 [Invoking Units], page 35), the value of '_' is not expanded:

```
You have: mile
You want: ft
        mile = 5280 ft
        mile = (1 / 0.00018939394) ft
You have: _
You want: m
        _ = 1609.344 m
        _ = (1 / 0.00062137119) m
```

You can give '_' at the '`You want:`' prompt, but it usually is not very useful.

5.6 Complicated Unit Expressions

The `units` program is especially helpful in ensuring accuracy and dimensional consistency when converting lengthy unit expressions. For example, one form of the Darcy–Weisbach fluid-flow equation is

$$\Delta P = \frac{8}{\pi^2}\rho f L \frac{Q^2}{d^5}$$

where ΔP is the pressure drop, ρ is the mass density, f is the (dimensionless) friction factor, L is the length of the pipe, Q is the volumetric flow rate, and d is the pipe diameter. You might want to have the equation in the form

$$\Delta P = A_1 \rho f L \frac{Q^2}{d^5}$$

that accepted the user's normal units; for typical units used in the US, the required conversion could be something like

```
You have: (8/pi^2)(lbm/ft^3)ft(ft^3/s)^2(1/in^5)
You want: psi
        * 43.533969
        / 0.022970568
```

The parentheses allow individual terms in the expression to be entered naturally, as they might be read from the formula. Alternatively, the multiplication could be done with the

'*' rather than a space; then parentheses are needed only around 'ft^3/s' because of its exponent:

```
You have: 8/pi^2 * lbm/ft^3 * ft * (ft^3/s)^2 /in^5
You want: psi
        * 43.533969
        / 0.022970568
```

Without parentheses, and using spaces for multiplication, the previous conversion would need to be entered as

```
You have: 8 lb ft ft^3 ft^3 / pi^2 ft^3 s^2 in^5
You want: psi
        * 43.533969
        / 0.022970568
```

5.7 Variables Assigned at Run Time

Unit definitions are fixed once `units` has finished reading the units data file(s), but at run time you can assign unit expressions to variables whose names begin with an underscore, using the syntax

```
_name = <unit expression>
```

This can help manage a long calculation by saving intermediate quantities as variables that you can use later. For example, to determine the shot-noise-limited signal-to-noise ratio (SNR) of an imaging system using a helium–neon laser, you could do

```
You have: _lambda = 632.8 nm            # laser wavelength
You have: _nu = c / _lambda             # optical frequency
You have: _photon_energy = h * _nu
You have: _power = 550 uW
You have: _photon_count = _power * 500 ns / _photon_energy
You have: _snr = sqrt(_photon_count)
You have: _snr
You want:
        Definition: sqrt(_photon_count) = 29597.922
```

Except for beginning with an underscore, runtime variables follow the same naming rules as units. Because names beginning with '_' are reserved for these variables and unit names cannot begin with '_', runtime variables can never hide unit definitions. Runtime variables are undefined until you make an assignment to them, so if you give a name beginning with an underscore and no assignment has been made, you get an error message.

When you assign a unit expression to a runtime variable, `units` checks the expression to determine whether it is valid, but the resulting definition is stored as a text string that is not reduced to primitive units. The text will be processed anew each time you use the variable in a conversion or calculation; this means that if your definition depends on other runtime variables (or the special variable '_'), the result of calculating with your variable will change if any of those variables change. A dependence need not be direct.

Continuing the example of the laser above, suppose you have done the calculation as shown. You now wonder what happens if you switch to an argon laser:

```
You have: _lambda = 454.6 nm
You have: _snr
You want:
        Definition: sqrt(_photon_count) = 25086.651
```

If you then change the power:

```
You have: _power = 1 mW
You have: _snr
You want:
        Definition: sqrt(_photon_count) = 33826.834
```

Instead of having to reenter or edit a lengthy expression when you perform another calculation, you need only enter values that change; in this respect, runtime variables are similar to a spreadsheet.

The more times a variable appears in an expression that depends on it, the greater the benefit of having a calculation using that expression reflect changes to that variable. For example, the length of daylight—the time the Sun is above the horizon—at a given latitude and declination of the Sun is given by

$$L = 2\cos^{-1}\left(\frac{\sin h - \sin\phi\sin\delta}{\cos\phi\cos\delta}\right)$$

where L is the day length, h is the Sun's altitude (elevation angle), ϕ is the location's latitude, and δ is the Sun's declination (angle with the equatorial plane). The result above is in sidereal time; the length in solar time is obtained by multiplying by

```
siderealday / day
```

By convention, the Sun's altitude at rise or set is $-50'$ to allow for atmospheric refraction and the semidiameter of its disk. At the summer solstice in the northern hemisphere, the Sun's declination is approximately 23.44°; to find the length of the longest day of the year for a latitude of 55°, you could do

```
You have: _alt = -50 arcmin
You have: _lat = 55 deg
You have: _decl = 23.44 deg
You have: _num = sin(_alt) - sin(_lat) sin(_decl)
You have: _denom = cos(_lat) cos(_decl)
You have: _sday = 2 (acos(_num / _denom) / circle) 24 hr
You have: _day = _sday siderealday / day
You have: _day
You want: hms
        17 hr + 19 min + 34.895151 sec
```

At the winter solstice, the Sun's declination is approximately −23.44°, so you could calculate the length of the shortest day of the year using:

```
You have: _decl = -23.44 deg
You have: _day
You want: hms
        7 hr + 8 min + 40.981084 sec
```

Latitude and declination each appear twice in the expression for `_day`; the result in the examples above is updated by changing only the value of the declination.

It may seem easier—and less subject to error—to simply specify the new value of `_decl` as the negative of the current value (e.g., '`_decl = -_decl`'). This doesn't work; when you make an assignment with the '`=`' operator, the definition is stored as entered, including possible dependencies on variables. But if you attempt an assignment that is ultimately self-referential, the current definition is retained, and you get an error message. For example,

```
You have: _decl = 23.44 deg
You have: _decl = -_decl
Circular unit definition
```

You can overcome this by using the '`:=`' operator, which reduces the right hand side to primitive units before making the assignment, eliminating any dependencies on variables. Returning to the example above,

```
You have: _decl = 23.44 deg
You have: _decl = -_decl
Circular unit definition
You have: _decl := -_decl
You have: _decl
You want: deg
        * -23.44
        / -0.042662116
```

This works to much the same effect as if the assignment had been entered literally, e.g.,

```
You have: _decl = -23.44 deg
```

but the actual definition is in primitive units—in this case, radians:

```
You have: _decl = 23.44 deg
You have: _decl := -_decl
You have: _decl
You want:
        Definition: -0.40910517666747087 radian = -0.40910518 radian
```

Definitions are text strings, and a redefinition using '`:=`' is given with enough digits maintain the full precision of the current definition when converted back to a number; because it is a string, all digits are displayed when showing the definition, regardless of the numerical display precision, so you may see more digits than expected.

A runtime variable must be assigned before it can be used in an assignment; in the first of the three examples above, giving the general equation before the values for `_alt`, `_lat`, and `_decl` had been assigned would result in an error message.

You can display a list of runtime variables you have assigned by typing *`search _`*.

5.8 Backwards Compatibility: '*' and '-'

The original `units` assigned multiplication a higher precedence than division using the slash. This differs from the usual precedence rules, which give multiplication and division equal precedence, and can be confusing for people who think of units as a calculator.

The star operator ('`*`') included in this `units` program has, by default, the same precedence as division, and hence follows the usual precedence rules. For backwards compatibility you can invoke `units` with the `--oldstar` option. Then '`*`' has a higher precedence than division, and the same precedence as multiplication using the space.

Historically, the hyphen ('`-`') has been used in technical publications to indicate products of units, and the original `units` program treated it as a multiplication operator. 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. For backwards compatibility use the `--product` option, which causes `units` to treat the binary '`-`' operator as a product operator. When '`-`' is a multiplication operator it has the same precedence as multiplication with a space, giving it a higher precedence than division.

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.

6 Nonlinear Unit Conversions

Nonlinear units are represented using functional notation. They make possible nonlinear unit conversions such as temperature.

6.1 Temperature Conversions

Conversions between temperatures are different from linear conversions between temperature *increments*—see the example below. The absolute temperature conversions are handled by units starting with '`temp`', and you must use functional notation. The temperature-increment conversions 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 that indicates that x should have units of '`tempF`' attached to it. See Section 14.3 [Defining Nonlinear Units], page 47. The first conversion shows that if it's 45 degrees Fahrenheit outside, it's 7.2 degrees Celsius. The second conversion indicates that a change of 45 degrees Fahrenheit corresponds to a change of 25 degrees Celsius. The conversion from '`tempF(x)`' is to absolute temperature, so that

```
You have: tempF(45)
You want: degR
        * 504.67
        / 0.0019814929
```

gives the same result as

```
You have: tempF(45)
You want: tempR
        * 504.67
        / 0.0019814929
```

But if you convert '`tempF(x)`' to '`degC`', the output is probably not what you expect:

```
You have: tempF(45)
You want: degC
        * 280.37222
        / 0.0035666871
```

The result is the temperature in K, because '`degC`' is defined as '`K`', the kelvin. For consistent results, use the '`tempX`' units when converting to a temperature rather than converting a temperature increment.

The '`tempC()`' and '`tempF()`' definitions are limited to positive absolute temperatures, and giving a value that would result in a negative absolute temperature generates an error message:

```
You have: tempC(-275)
                    ^

Argument of function outside domain
```

6.2 US Consumer Price Index

`units` includes the US Consumer Price Index published by the US Bureau of Labor Statistics. Several functions that use this value are provided: '`cpi`', '`cpi_now`', '`inflation_since`', and '`dollars_in`'.

The '`cpi`' function gives the CPI for a specified decimal year. A *decimal year* is given as the year plus the fractional part of the year; because of leap years and the different lengths of months, calculating an exact value for the fractional part can be tedious, but for the purposes of CPI, an approximate value is usually adequate. For example, 1 January 2000 is 2000.0, 1 April 2000 is 2000.25, 1 July 2000 is 2000.4986, and 1 October 2000 is 2000.75. Note also that the CPI data update monthly; values in between months are linearly interpolated.

In the middle of 1975, the CPI was

```
You have: cpi(1975.5)
You want:
        Definition: 53.6
```

The value of the CPI for a month is usually published sometime around the 20th day of the following month; the latest value of the CPI is available with '`cpi_now`'. On 7 January 2024, the value was

```
You have: cpi_now
You want:
        Definition: UScpi_now = 307.051
```

This means that the CPI was 307.015 on 1 December 2023. The '`cpi_now`' variable can only present the most recent data available, so it can lag the current CPI by several weeks. The decimal year of the last update is available with '`cpi_lastdate`'.

The ‘inflation_since’ function provides a convenient way to determine the inflation factor from a specified decimal year to the latest value in the CPI table. For example, on 7 January 2024:

```
You have: inflation_since(1970)
You want:
        Definition: 8.1445889
```

In other words, goods that cost 1 US$ in 1970 would cost 8.14 US$ on 1 December 2023.

The ‘inflation_since’ function can be used to determine an annual rate of inflation. The earliest US CPI data are from about 1913.1; the approximate time between then and 7 January 2024 is 110.9 years. The approximate annual inflation rate for that period is then

```
You have: inflation_since(1913.1)^1|110.9 - 1
You want: %
        * 3.1548115
        / 0.31697614
```

The inflation rate for any time period can be found from the ratio of the CPI at the end of the period to that of the beginning:

```
You have: (cpi(1982)/cpi(1972))^1|10 - 1
You want: %
        * 8.6247033
        / 0.11594602
```

The period 1972–1982 was indeed one of high inflation.

The ‘dollars_in’ function is similar to ‘inflation_since’ but its output is in US$ rather than dimensionless:

```
You have: dollars_in(1970)
You want:
        Definition: 8.1445889 US$
```

A typical use might be

```
You have: 250 dollars_in(1970)
You want: $
        * 2036.1472
        / 0.00049112362
```

Because ‘dollars_in’ includes the units, you should not include them at the ‘You have:’ prompt. You can also use ‘dollars_in’ to convert between two specified years:

```
You have: 250 dollars_in(1970)
You want: dollars_in(1950)
        * 156.49867
        / 0.0063898305
```

which shows that 250 US$ in 1970 would have equivalent purchasing power to 156 US$ in 1950.

6.3 Other Nonlinear Units

Some other examples of nonlinear units are numerous different ring sizes and wire gauges, screw gauges, pipe and tubing sizes, the grit sizes used for abrasives, the decibel scale, shoe size, scales for the density of sugar (e.g., baume). The standard data file also supplies units for computing the area of a circle and the volume of a sphere. See the standard units data file for more details.

Diameters of American wire sizes can be found using the '`wiregauge()`' function or its alias '`awg()`':

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

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

Wire and screw gauges with multiple zeroes are signified using negative numbers, where two zeroes ("00"; "2/0") is '`-1`', three zeros ("000"; "3/0") is '`-2`', and so on. Alternatively, you can use the synonyms '`g00`', '`g000`', or '`g2_0`', '`g3_0`', and so on that are defined in the standard units data file.

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

In North America, wire sizes larger than 0000 ("4/0") are usually given in terms of area, either in kcmil or the older initialism MCM (thousand circular mils). Outside of North America, all wire sizes are usually given in terms of area in mm^2. Wire area can be obtained using '`wiregaugeA()`' or its alias '`awgA()`':

```
You have: awgA(g6_0)
You want: kcmil
        * 336.45718
        / 0.0029721464
You have: awgA(12)
You want: mm^2
        * 3.3087729
        / 0.30222685
```

The closest standard metric sizes are 2.5 mm^2 and 4 mm^2; in general, there isn't an exact correlation between American and metric wire sizes.

Though based on the long-established iron pipe size (IPS) given in inches, nominal pipe size (NPS) is a dimensionless quantity that corresponds to the inch size. Pipe size can be equivalently specified using metric diamètre nominal (DN), which roughly corresponds to the diameter in mm. For a given pipe size, outside diameter is constant while inside diameter varies with schedule. For example, for NPS $2\frac{1}{2}$ pipe,

```
You have: npsOD(2+1|2)
You want: in
        * 2.875
        / 0.34782609
You have: nps40(2+1|2)
You want: in
        * 2.469
        / 0.40502228
You have: nps80(2+1|2)
You want: in
        * 2.323
        / 0.43047783
```

Pipe size can be given equivalently in terms of the metric DN by using the 'DN()' function, which converts nominal metric size to nominal inch size:

```
You have: npsOD(DN(65))
You want: mm
        * 73.025
        / 0.01369394
You have: _
You want: in
        * 2.875
        / 0.34782609
```

Unlike with wire sizes, actual NPS and metric DN pipe dimensions are the same.

```
You have: grit_P(600)
You want: grit_ansicoated
        342.76923
```

The last example shows the conversion from P graded sand paper, which is the European standard and may be marked "P600" on the back, to the USA standard.

You can compute the area of a circle using the nonlinear unit, 'circlearea'. You can also do this using the circularinch or circleinch. The next example shows two ways to compute the area of a circle with a five inch radius and one way to compute the volume of a sphere with a radius of one meter.

```
You have: circlearea(5 in)
You want: in2
        * 78.539816
        / 0.012732395

You have: 10^2 circleinch
You want: in2
        * 78.539816
        / 0.012732395

You have: spherevol(meter)
You want: ft3
        * 147.92573
        / 0.0067601492
```

The inverse of a nonlinear conversion is indicated by prefixing a tilde ('~') to the nonlinear unit name:

```
You have: ~wiregauge(0.090742002 inches)
You want:
        Definition: 11
```

You can give a nonlinear unit definition without an argument or parentheses, and press `Enter` at the '`You want:`' prompt to get the definition of a nonlinear unit; if the definition is not valid for all real numbers, the range of validity is also given. If the definition requires specific units this information is also displayed:

```
You have: tempC
        Definition: tempC(x) = x K + stdtemp
                    defined for x >= -273.15
You have: ~tempC
        Definition: ~tempC(tempC) = (tempC +(-stdtemp))/K
                    defined for tempC >= 0 K
You have: circlearea
        Definition: circlearea(r) = pi r^2
                    r has units m
```

To see the definition of the inverse use the '~' notation. In this case the parameter in the functional definition will usually be the name of the unit. Note that the inverse for '`tempC`' shows that it requires units of '`K`' in the specification of the allowed range of values. Nonlinear unit conversions are described in more detail in Section 14.3 [Defining Nonlinear Units], page 47.

6.4 Multivariate Functions

The `units` program supports some multivariate functions for performing calculations with units. The '`hypot`' function computes the length of a hypotenuse given two sides. It works with any pair of values that have compatible dimensions.

```
You have: hypot(3 m, 4 m)
You want:
        Definition: 5 m
```

You can compute wind chill, based on temperature and wind speed, or heat index, based on temperature and humidity, with humidity given as a fraction, e.g., 0.5 for a 50% relative humidity.

```
You have: windchill(tempF(28), 14 mph)
You want: tempF
        16.86826
You have: heatindex(tempC(40),0.5)
You want: tempC
        54.767952
```

Both wind chill and heat index return a value that is nominally a temperature, so you will want to convert it to Fahrenheit or Celsius as shown in order to get a meaningful numerical value.

7 Unit Lists: Conversion to Sums of Units

Outside of the SI, it is sometimes desirable to convert a single unit to a sum of units—for example, feet to feet plus inches. The conversion *from* sums of units was described in Section 5.2 [Sums and Differences of Units], page 11, and is a simple matter of adding the units with the '+' sign:

```
You have: 12 ft + 3 in + 3|8 in
You want: ft
        * 12.28125
        / 0.081424936
```

Although you can similarly write a sum of units to convert *to*, the result will not be the conversion to the units in the sum, but rather the conversion to the particular sum that you have entered:

```
You have: 12.28125 ft
You want: ft + in + 1|8 in
        * 11.228571
        / 0.089058524
```

The unit expression given at the '`You want:`' prompt is equivalent to asking for conversion to multiples of '`1 ft + 1 in + 1|8 in`', which is 1.09375 ft, so the conversion in the previous example is equivalent to

```
You have: 12.28125 ft
You want: 1.09375 ft
        * 11.228571
        / 0.089058524
```

In converting to a sum of units like miles, feet and inches, you typically want the largest integral value for the first unit, followed by the largest integral value for the next, and the remainder converted to the last unit. You can do this conversion easily with `units` using a special syntax for lists of units. You must list the desired units in order from largest to smallest, separated by the semicolon ('`;`') character:

```
You have: 12.28125 ft
You want: ft;in;1|8 in
        12 ft + 3 in + 3|8 in
```

The conversion always gives integer coefficients on the units in the list, except possibly the last unit when the conversion is not exact:

```
You have: 12.28126 ft
You want: ft;in;1|8 in
        12 ft + 3 in + 3.00096 * 1|8 in
```

The order in which you list the units is important:

```
You have: 3 kg
You want: oz;lb
        105 oz + 0.051367866 lb

You have: 3 kg
You want: lb;oz
        6 lb + 9.8218858 oz
```

Listing ounces before pounds produces a technically correct result, but not a very useful one. You must list the units in descending order of size in order to get the most useful result.

Ending a unit list with the separator ';' has the same effect as repeating the last unit on the list, so 'ft;in;1|8 in;' is equivalent to 'ft;in;1|8 in;1|8 in'. With the example above, this gives

```
You have: 12.28126 ft
You want: ft;in;1|8 in;
        12 ft + 3 in + 3|8 in + 0.00096 * 1|8 in
```

in effect separating the integer and fractional parts of the coefficient for the last unit. If you instead prefer to round the last coefficient to an integer you can do this with the `--round` (`-r`) option. With the previous example, the result is

```
You have: 12.28126 ft
You want: ft;in;1|8 in
        12 ft + 3 in + 3|8 in (rounded down to nearest 1|8 in)
```

When you use the `-r` option, repeating the last unit on the list has no effect (e.g., 'ft;in;1|8 in;1|8 in' is equivalent to 'ft;in;1|8 in'), and hence neither does ending a list with a ';'. With a single unit and the `-r` option, a terminal ';' *does* have an effect: it causes `units` to treat the single unit as a list and produce a rounded value for the single unit. Without the extra ';', the `-r` option has no effect on single unit conversions. This example shows the output using the `-r` option:

```
You have: 12.28126 ft
You want: in
        * 147.37512
        / 0.0067854058

You have: 12.28126 ft
You want: in;
        147 in (rounded down to nearest in)
```

Each unit that appears in the list must be conformable with the first unit on the list, and of course the listed units must also be conformable with the unit that you enter at the 'You have:' prompt.

```
You have: meter
You want: ft;kg
             ^
conformability error
        ft = 0.3048 m
        kg = 1 kg

You have: meter
You want: lb;oz
conformability error
        1 m
        0.45359237 kg
```

In the first case, `units` reports the disagreement between units appearing on the list. In the second case, `units` reports disagreement between the unit you entered and the desired conversion. This conformability error is based on the first unit on the unit list.

Other common candidates for conversion to sums of units are angles and time:

```
You have: 23.437754 deg
You want: deg;arcmin;arcsec
    23 deg + 26 arcmin + 15.9144 arcsec

You have: 7.2319 hr
You want: hr;min;sec
    7 hr + 13 min + 54.84 sec
```

Some applications for unit lists may be less obvious. Suppose that you have a postal scale and wish to ensure that it's accurate at 1 oz, but have only metric calibration weights. You might try

```
You have: 1 oz
You want: 100 g;50 g; 20 g;10 g;5 g;2 g;1 g;
        20 g + 5 g + 2 g + 1 g + 0.34952312 * 1 g
```

You might then place one each of the 20 g, 5 g, 2 g, and 1 g weights on the scale and hope that it indicates close to

```
You have: 20 g + 5 g + 2 g + 1 g
You want: oz;
        0.98767093 oz
```

Appending '`;`' to '`oz`' forces a one-line display that includes the unit; here the integer part of the result is zero, so it is not displayed.

If a non-empty list item differs vastly in scale from the quantity from which the list is to be converted, you may exceed the available precision of floating point (about 15 digits), in which case you will get a warning, e.g.,

```
You have: lightyear
You want: mile;100 inch;10 inch;mm;micron
        5.8786254e+12 mile + 390 * 100 inch (at 15-digit precision limit)
```

7.1 Cooking Measure

In North America, recipes for cooking typically measure ingredients by volume, and use units that are not always convenient multiples of each other. Suppose that you have a recipe for 6 and you wish to make a portion for 1. If the recipe calls for 2 1/2 cups of an ingredient, you might wish to know the measurements in terms of measuring devices you have available, you could use `units` and enter

```
You have: (2+1|2) cup / 6
You want: cup;1|2 cup;1|3 cup;1|4 cup;tbsp;tsp;1|2 tsp;1|4 tsp
        1|3 cup + 1 tbsp + 1 tsp
```

By default, if a unit in a list begins with fraction of the form 1|*x* and its multiplier is an integer, the fraction is given as the product of the multiplier and the numerator; for example,

```
You have: 12.28125 ft
You want: ft;in;1|8 in;
        12 ft + 3 in + 3|8 in
```

In many cases, such as the example above, this is what is wanted, but sometimes it is not. For example, a cooking recipe for 6 might call for 5 1/4 cup of an ingredient, but you want a portion for 2, and your 1-cup measure is not available; you might try

```
You have: (5+1|4) cup / 3
You want: 1|2 cup;1|3 cup;1|4 cup
        3|2 cup + 1|4 cup
```

This result might be fine for a baker who has a 1 1/2-cup measure (and recognizes the equivalence), but it may not be as useful to someone with more limited set of measures, who does want to do additional calculations, and only wants to know "How many 1/2-cup measures to I need to add?" After all, that's what was actually asked. With the `--show-factor` option, the factor will not be combined with a unity numerator, so that you get

```
You have: (5+1|4) cup / 3
You want: 1|2 cup;1|3 cup;1|4 cup
        3 * 1|2 cup + 1|4 cup
```

A user-specified fractional unit with a numerator other than 1 is never overridden, however—if a unit list specifies '`3|4 cup;1|2 cup`', a result equivalent to 1 1/2 cups will always be shown as '`2 * 3|4 cup`' whether or not the `--show-factor` option is given.

7.2 Unit List Aliases

A unit list such as

```
cup;1|2 cup;1|3 cup;1|4 cup;tbsp;tsp;1|2 tsp;1|4 tsp
```

can be tedious to enter. The `units` program provides shorthand names for some common combinations:

`hms`	time: hours, minutes, seconds
`dms`	angle: degrees, minutes, seconds
`time`	time: years, days, hours, minutes and seconds
`usvol`	US cooking volume: cups and smaller
`uswt`	US weight: pounds and ounces
`ftin`	length: feet, inches and 1/8 inches
`ftin2`	length: feet, inches and 1/2 inches
`ftin4`	length: feet, inches and 1/4 inches
`ftin8`	length: feet, inches and 1/8 inches
`ftin16`	length: feet, inches and 1/16 inches
`ftin32`	length: feet, inches and 1/32 inches
`ftin64`	length: feet, inches and 1/64 inches
`inchfine`	length: inches subdivided to 1/64 inch

Using these shorthands, or *unit list aliases*, you can do the following conversions:

```
You have: anomalisticyear
You want: time
        1 year + 25 min + 3.4653216 sec
You have: 1|6 cup
You want: usvol
        2 tbsp + 2 tsp
```

Suppose you want to drill a clearance hole for a #10 screw and have about 1/64 inch clearance; you could try

```
You have: screwgauge(10) + 1|64 in
You want: ftin64
        13.16 * 1|64 in
You have: _
You want: ftin32
        6.58 * 1|32 in
```

If a slightly tight fit is acceptable, a 13/64-inch drill would do the job; if not, a 7/32-inch drill would work with a slightly looser fit.

You can define your own unit list aliases; see Section 14.6 [Defining Unit List Aliases], page 52.

You cannot combine a unit list alias with other units: it must appear alone at the '`You want:`' prompt.

You can display the definition of a unit list alias by entering it at the '`You have:`' prompt:

```
You have: dms
        Definition: unit list, deg;arcmin;arcsec
```

When you specify compact output with `--compact`, `--terse` or `-t` and perform conversion to a unit list, `units` lists the conversion factors for each unit in the list, separated by semicolons.

```
You have: year
You want: day;min;sec
365;348;45.974678
```

Unlike the case of regular output, zeros *are* included in this output list:

```
You have: liter
You want: cup;1|2 cup;1|4 cup;tbsp
4;0;0;3.6280454
```

8 Alternative Unit Systems

8.1 CGS Units

The SI—an extension of the MKS (meter–kilogram–second) system—has largely supplanted the older CGS (centimeter–gram–second) system, but CGS units are still used in a few specialized fields, especially in physics where they lead to a more elegant formulation of Maxwell's equations. Conversions between SI and CGS involving mechanical units are straightforward, involving powers of 10 (e.g., 1 m = 100 cm). Conversions involving electromagnetic units are more complicated, and `units` supports four different systems of CGS units: electrostatic units (ESU), electromagnetic units (EMU), the Gaussian system and the Heaviside–Lorentz system. The differences between these systems arise from different choices made for proportionality constants in electromagnetic equations. Coulomb's law gives electrostatic force between two charges separated by a distance r:

$$F = k_{\mathrm{C}} \frac{q_1 q_2}{r^2}.$$

Ampere's law gives the electromagnetic force per unit length between two current-carrying conductors separated by a distance r:

$$\frac{F}{\ell} = 2k_{\mathrm{A}} \frac{I_1 I_2}{r}.$$

The two constants, k_{C} and k_{A}, are related by the square of the speed of light:

$$k_{\mathrm{A}} = k_{\mathrm{C}}/c^2.$$

In the SI, the constants have dimensions, and an additional base unit, the ampere, measures electric current. The CGS systems do not define new base units, but express charge and current as derived units in terms of mass, length, and time. In the ESU system, the constant for Coulomb's law is chosen to be unity and dimensionless, which defines the unit of charge. In the EMU system, the constant for Ampere's law is chosen to be unity and dimensionless, which defines a unit of current. The Gaussian system usually uses the ESU units for charge and current; it chooses another constant so that the units for the electric and magnetic fields are the same. The Heaviside–Lorentz system is "rationalized" so that factors of 4π do not appear in Maxwell's equations. The SI system is similarly rationalized, but the other CGS systems are not. In the Heaviside–Lorentz (HLU) system the factor of 4π appears in Coulomb's law instead; this system differs from the Gaussian system by factors of $\sqrt{4\pi}$.

The dimensions of electrical quantities in the various CGS systems are different from the SI dimensions for the same units; strictly, conversions between these systems and SI are not possible. But units in different systems relate to the same physical quantities, so there is a *correspondence* between these units. The `units` program defines the units so that you can convert between corresponding units in the various systems.

8.1.1 Specifying CGS Units

The CGS definitions involve $\mathrm{cm}^{1/2}$ and $\mathrm{g}^{1/2}$, which is problematic because `units` does not normally support fractional roots of base units. The `--units` (`-u`) option allows selection of a CGS unit system and works around this restriction by introducing base units for the square roots of length and mass: '`sqrt_cm`' and '`sqrt_g`'. The centimeter then becomes '`sqrt_cm^2`' and the gram, '`sqrt_g^2`'. This allows working from equations using the units in the CGS system, and enforcing dimensional conformity within that system. Recognized CGS arguments to the `--units` option are '`gauss[ian]`', '`esu`', '`emu`', '`lhu`'; the argument is case insensitive. You can also give '`si`' which just enforces the default SI mode and displays '`(SI)`' at the '`You have:`' prompt to emphasize the units mode. Some other types of units are also supported as described below. Giving an unrecognized system generates a warning, and `units` uses SI units.

The changes resulting from the `--units` option are actually controlled by the `UNITS_SYSTEM` environment variable. If you frequently work with one of the supported CGS units systems, you may set this environment variable rather than giving the `--units` option at each invocation. As usual, an option given on the command line overrides the setting of the environment variable. For example, if you would normally work with Gaussian units but might occasionally work with SI, you could set `UNITS_SYSTEM` to '`gaussian`' and specify SI with the `--units` option. Unlike the argument to the `--units` option, the value of `UNITS_SYSTEM` *is* case sensitive, so setting a value of '`EMU`' will have no effect other than to give an error message and set SI units.

The CGS definitions appear as conditional settings in the standard units data file, which you can consult for more information on how these units are defined, or on how to define an alternate units system.

8.1.2 CGS Units Systems

The ESU system derives the electromagnetic units from its unit of charge, the statcoulomb, which is defined from Coulomb's law. The statcoulomb equals $\mathrm{dyne}^{1/2}\,\mathrm{cm}$ or $\mathrm{cm}^{3/2}\,\mathrm{g}^{1/2}\,\mathrm{s}^{-1}$. The unit of current, the statampere, is statcoulomb sec, analogous to the relationship in SI. Other electrical units are then derived in a manner similar to that for SI units; the units use the SI names prefixed by '`stat-`', e.g., '`statvolt`' or '`statV`'. The prefix '`st-`' is also recognized (e.g., '`stV`').

The EMU system derives the electromagnetic units from its unit of current, the abampere, which is defined in terms of Ampere's law. The abampere is equal to $\mathrm{dyne}^{1/2}$ or $\mathrm{cm}^{1/2}\mathrm{g}^{1/2}\mathrm{s}^{-1}$. The unit of charge, the abcoulomb, is abampere sec, again analogous to the SI relationship. Other electrical units are then derived in a manner similar to that for SI units; the units use the SI names prefixed by '`ab-`', e.g., '`abvolt`' or '`abV`'. The magnetic field units include the gauss, the oersted and the maxwell.

The Gaussian units system, which was also known as the Symmetric System, uses the same charge and current units as the ESU system (e.g., '`statC`', '`statA`'); it differs by defining the magnetic field so that it has the same units as the electric field. The resulting magnetic field units are the same ones used in the EMU system: the gauss, the oersted and the maxwell.

The Heaviside–Lorentz system appears to lack named units. We define five basic units, '`hlu_charge`', '`hlu_current`', '`hlu_volt`', '`hlu_efield`' and '`hlu_bfield`' for conversions with this system. It is important to remember that with all of the CGS systems, the units may look the same but mean something different. The HLU system and Gaussian systems both measure magnetic field using the same CGS dimensions, but the amount of magnetic field with the same units is different in the two systems.

8.1.3 Conversions Between Different Systems

The CGS systems define units that measure the same thing but may have conflicting dimensions. Furthermore, the dimensions of the electromagnetic CGS units are never compatible with SI. But if you measure charge in two different systems you have measured the same physical thing, so there is a *correspondence* between the units in the different systems, and `units` supports conversions between corresponding units. When running with SI, `units` defines all of the CGS units in terms of SI. When you select a CGS system, `units` defines the SI units and the other CGS system units in terms of the system you have selected.

```
(Gaussian) You have: statA
           You want: abA
        * 3.335641e-11
        / 2.9979246e+10
(Gaussian) You have: abA
           You want: sqrt(dyne)
conformability error
        2.9979246e+10 sqrt_cm^3 sqrt_g / s^2
        1 sqrt_cm sqrt_g / s
```

In the above example, `units` converts between the current units statA and abA even though the abA, from the EMU system, has incompatible dimensions. This works because in Gaussian mode, the abA is defined in terms of the statA, so it does not have the correct definition for EMU; consequently, you cannot convert the abA to its EMU definition.

One challenge of conversion is that because the CGS system has fewer base units, quantities that have different dimensions in SI may have the same dimension in a CGS system. And yet, they may not have the same conversion factor. For example, the unit for the E field and B fields are the same in the Gaussian system, but the conversion factors to SI are quite different. This means that correct conversion is only possible if you keep track of what quantity is being measured. You cannot convert statV/cm to SI without indicating which type of field the unit measures. To aid in dimensional analysis, `units` defines various dimension units such as '`LENGTH`', '`TIME`', and '`CHARGE`' to be the appropriate dimension in SI. The electromagnetic dimensions such as '`B_FIELD`' or '`E_FIELD`' may be useful aids both for conversion and dimensional analysis in CGS. You can convert them to or from CGS in order to perform SI conversions that in some cases will not work directly due to dimensional incompatibilities. This example shows how the Gaussian system uses the same units for all of the fields, but they all have different conversion factors with SI.

```
(Gaussian) You have: statV/cm
           You want: E_FIELD
        * 29979.246
        / 3.335641e-05
(Gaussian) You have: statV/cm
           You want: B_FIELD
        * 0.0001
        / 10000
(Gaussian) You have: statV/cm
           You want: H_FIELD
        * 79.577472
        / 0.012566371
(Gaussian) You have: statV/cm
           You want: D_FIELD
        * 2.6544187e-07
        / 3767303.1
```

The next example shows that the oersted cannot be converted directly to the SI unit of magnetic field, A/m, because the dimensions conflict. We cannot redefine the ampere to make this work because then it would not convert with the statampere. But you can still do this conversion as shown below.

```
(Gaussian) You have: oersted
           You want: A/m
conformability error
        1 sqrt_g / s sqrt_cm
        29979246 sqrt_cm sqrt_g / s^2
(Gaussian) You have: oersted
           You want: H_FIELD
        * 79.577472
        / 0.012566371
```

8.2 Natural Units

Like the CGS units, “natural” units are an alternative to the SI system used primarily physicists in different fields, with different systems tailored to different fields of study. These systems are “natural” because the base measurements are defined using physical constants instead of arbitrary values such as the meter or second. In different branches of physics, different physical constants are more fundamental, which has given rise to a variety of incompatible natural unit systems.

The supported systems are the “natural” units (which seem to have no better name) used in high energy physics and cosmology, the Planck units, often used by scientists working with gravity, and the Hartree atomic units are favored by those working in physical chemistry and condensed matter physics.

You can select the various natural units using the `--units` option in the same way that you select the CGS units. The “natural” units come in two types, a rationalized system derived from the Heaviside–Lorentz units and an unrationalized system derived from the Gaussian system. You can select these using ‘`natural`’ and ‘`natural-gauss`’ respectively. For conversions in SI mode, several unit names starting with ‘`natural`’ are available. This “natural” system is defined by setting $\hbar$, c and the Boltzman constant to 1. Only a single base unit remains: the electron volt.

The Planck units exist in a variety of forms, and `units` supports two. Both supported forms are rationalized, in that factors of 4π do not appear in Maxwell’s equations. However, Planck units can also differ based on how the gravitational constant is treated. This system is similar to the natural units in that c, $\hbar$, and Boltzman’s constant are set to 1, but in this system, Newton’s gravitational constant, G, is also fixed. In the “reduced” Planck system, $8\pi G = 1$ whereas in the unreduced system $G = 1$. The reduced system eliminates factors of 8π from the Einstein field equations for gravitation, so this is similar to the process of forming rationalized units to simplify Maxwell’s equations. To obtain the unreduced system use the name ‘`planck`’ and for the reduced Planck units, ‘`planck-red`’. Units such as ‘`planckenergy`’ and ‘`planckenergy_red`’ enable you to convert the unreduced and reduced Planck energy unit in SI mode between the various systems. In Planck units, all measurements are dimensionless.

The final natural unit system is the Hartree atomic units. Like the Planck units, all measurements in the Hartree units are dimensionless, but this system is defined by defined from completely different physical constants: the electron mass, Planck’s constant, the electron charge, and the Coulomb constant are the defining physical quantities, which are all set to unity. To invoke this system with the `--units` option use the name ‘`hartree`’.

8.3 Prompt Prefix

If a unit system is specified with the `--units` option, the selected system's name is prepended to the '`You have:`' prompt as a reminder, e.g.,

```
(Gaussian) You have: stC
            You want:
         Definition: statcoulomb = sqrt(dyne) cm = 1 sqrt_cm^3 sqrt_g / s
```

You can suppressed the prefix by including a line

```
!prompt
```

with no argument in a site or personal units data file. The prompt can be conditionally suppressed by including such a line within '`!var`' ... '`!endvar`' constructs, e.g.,

```
!var UNITS_SYSTEM gaussian gauss
!prompt
!endvar
```

This might be appropriate if you normally use Gaussian units and find the prefix distracting but want to be reminded when you have selected a different CGS system.

9 Logging Calculations

The `--log` option allows you to save the results of calculations in a file; this can be useful if you need a permanent record of your work. For example, the fluid-flow conversion in Section 5.6 [Complicated Unit Expressions], page 15, is lengthy, and if you were to use it in designing a piping system, you might want a record of it for the project file. If the interactive session

```
# Conversion factor A1 for pressure drop
# dP = A1 rho f L Q^2/d^5
You have: (8/pi^2) (lbm/ft^3)ft(ft^3/s)^2(1/in^5) # Input units
You want: psi
        * 43.533969
        / 0.022970568
```

were logged, the log file would contain

```
### Log started Fri Oct 02 15:55:35 2015

# Conversion factor A1 for pressure drop
# dP = A1 rho f L Q^2/d^5
From: (8/pi^2) (lbm/ft^3)ft(ft^3/s)^2(1/in^5)   # Input units
To:   psi
        * 43.533969
        / 0.022970568
```

The time is written to the log file when the file is opened.

The use of comments can help clarify the meaning of calculations for the log. The log includes conformability errors between the units at the '`You have:`' and '`You want:`' prompts, but not other errors, including lack of conformability of items in sums or differences or among items in a unit list. For example, a conversion between zenith angle and elevation angle could involve

```
You have: 90 deg - (5 deg + 22 min + 9 sec)
                                   ^
Invalid sum or difference of non-conformable units
You have: 90 deg - (5 deg + 22 arcmin + 9 arcsec)
You want: dms
        84 deg + 37 arcmin + 51 arcsec
You have: _
You want: deg
        * 84.630833
        / 0.011816024
You have:
```

The log file would contain

```
From: 90 deg - (5 deg + 22 arcmin + 9 arcsec)
To:   deg;arcmin;arcsec
        84 deg + 37 arcmin + 51 arcsec
From: _
To:   deg
        * 84.630833
        / 0.011816024
```

The initial entry error (forgetting that minutes have dimension of time, and that arcminutes must be used for dimensions of angle) does not appear in the output. When converting to a unit list alias, `units` expands the alias in the log file.

The '`From:`' and '`To:`' tags are written to the log file even if the `--quiet` option is given. If the log file exists when `units` is invoked, the new results are appended to the log file. The time is written to the log file each time the file is opened. The `--log` option is ignored when `units` is used non-interactively.

10 Invoking `units`

You invoke `units` like this:

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

If the *from-unit* and *to-unit* are omitted, 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 may need to be quoted to protect them from shell interpretation and to group them into two arguments. Note also that the `--quiet` option is enabled by default if you specify *from-unit* on the command line. See Chapter 3 [Command Line Use], page 4.

The default behavior of `units` can be changed by various options given on the command line. In most cases, the options may be given in either short form (a single '`-`' followed by a single character) or long form (`--` followed by a word or hyphen-separated words). Short-form options are cryptic but require less typing; long-form options require more typing but are more explanatory and may be more mnemonic. With long-form options you need only enter sufficient characters to uniquely identify the option to the program. For example, `--out %f` works, but `--o %f` fails because `units` has other long options beginning with ‘`o`’. However, `--q` works because `--quiet` is the only long option beginning with ‘`q`’.

Some options require arguments to specify a value (e.g., `-d 12` or `--digits 12`). Short-form options that do not take arguments may be concatenated (e.g., `-erS` is equivalent to `-e -r -S`); the last option in such a list may be one that takes an argument (e.g., `-ed 12`). With short-form options, the space between an option and its argument is optional (e.g., `-d12` is equivalent to `-d 12`). Long-form options may not be concatenated, and the space between a long-form option and its argument is required. Short-form and long-form options may be intermixed on the command line. Options may be given in any order, but when incompatible options (e.g., `--output-format` and `--exponential`) are given in combination, behavior is controlled by the last option given. For example, `-o%.12f -e` gives exponential format with the default eight significant digits).

Many options can be set interactively; this can be especially helpful for Windows users who start `units` from a shortcut. See Chapter 11 [Setting Options Interactively], page 41, for more information.

The following options are available:

`-c`
`--check` Check that all units and prefixes defined in units data files reduce to primitive units. Display a list of all units that cannot be reduced and a list of units with circular definitions. Also display some other diagnostics about suspicious definitions in the units data file. Only definitions active in the current locale are checked. You should always run `units` with this option after modifying a units data file.

Some errors may hide other errors, so you should run `units` with this option again after correcting any errors, and keep doing so until there are no errors.

`--check-verbose`
`--verbose-check`
Like the `--check` option, this option displays a list of units that cannot be reduced. But it also lists the units as they are checked. Because the `--check` option now catches circular unit definitions that previously caused `units` to hang, this option is no longer necessary. It is retained only for compatibility with previous versions.

`--list-units`
This option displays a list of all units and exits. With output redirected to a file, it can produce a list of all units in the database, which may comprise multiple `units` data files. It can also be useful if you aren’t sure of the name of a unit or want to find all units of a particular type and you want a more advanced search than is available with the interactive *`search`* command. See Section 3.1 [Finding Units], page 5, for some examples of how this option can be used.

`-d` *ndigits*
`--digits` *ndigits*

Set the number of significant digits in the output to the value specified (which must be greater than zero). For example, `-d 12` sets the number of significant digits to 12. With exponential output, `units` displays one digit to the left of the decimal point and eleven digits to the right of the decimal point. On most systems, the maximum number of internally meaningful digits is 15; if you specify a greater number than your system's maximum, `units` will print a warning and set the number to the largest meaningful value. To directly set the maximum value, give an argument of `max` (e.g., `-d max`). Be aware, of course, that "significant" here refers only to the *display* of numbers; if results depend on physical constants not known to this precision, the physically meaningful precision may be less than that shown. The `--digits` option is incompatible with the `--output-format` option; if you give them both, the format is controlled by the last option given.

`-e`
`--exponential`

Set the numeric output format to exponential (i.e., scientific notation), like that used in the Unix `units` program. The default precision is eight significant digits (seven digits to the right of the decimal point); this can be changed with the `--digits` option. The `--exponential` option is incompatible with the `--output-format` option; if you give them both, the format is controlled by the last option given.

`-o` *format*
`--output-format` *format*

This option affords complete control over the numeric output format using the specified *format*. The format is a single floating point numeric format for the `printf` function in the C programming language. All compilers support the format types '`g`' and '`G`' to specify significant digits, '`e`' and '`E`' for scientific notation, and '`f`' for fixed-point decimal. The ISO C99 standard introduced the '`F`' type for fixed-point decimal and the '`a`' and '`A`' types for hexadecimal floating point; these types are allowed with compilers that support them. The default format is '`%.8g`'; for greater precision, you could specify `-o %.15g`. Unlike with the `--digits` option, you can specify any desired precision, though not all digits may be meaningful. See Chapter 15 [Numeric Output Format], page 53, and the documentation for `printf` for more detailed descriptions of the format specification. The `--output-format` option affords the greatest control of the output appearance, but requires at least rudimentary knowledge of the `printf` format syntax. If you don't want to bother with the `printf` syntax, you can specify greater precision more simply with the `--digits` option or select exponential format with `--exponential`. The `--output-format` option is incompatible with the `--exponential` and `--digits` options; if you give either in combination with `--output-format`, the format is controlled by the last option given.

`-f filename`
`--file filename`
Instruct `units` to load the units file *filename*. You can specify up to 25 units files on the command line. When you use this option, `units` will load *only* the files you list on the command line; it will not load the standard file or your personal units file unless you explicitly list them. If *filename* is the empty string (`-f ""`), the default main units file (or that specified by `UNITSFILE`) will be loaded in addition to any others specified with `-f`.

`-L logfile`
`--log logfile`
Save the results of calculations in the file *logfile*; this can be useful if it is important to have a record of unit conversions or other calculations that are to be used extensively or in a critical activity such as a program or design project. If *logfile* exits, the new results are appended to the file. This option is ignored when `units` is used non-interactively. See Chapter 9 [Logging Calculations], page 34, for a more detailed description and some examples.

`-H filename`
`--history filename`
Instruct `units` to save history to *filename*, so that a record of your commands is available for retrieval across different `units` invocations. To prevent the history from being saved set *filename* to the empty string (`-H ""`). This option has no effect if readline is not available.

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

`-m`
`--minus` Causes '`-`' to be interpreted as a subtraction operator. This is the default behavior.

`-p`
`--product`
Causes '`-`' to be interpreted as a multiplication operator when it has two operands. It will act as a negation operator when it has only one operand: '`(-3)`'. By default '`-`' is treated as a subtraction operator.

`--oldstar`
Causes '`*`' to have the old-style precedence, higher than the precedence of division so that '`1/2*3`' will equal '`1/6`'.

`--newstar`
Forces '`*`' to have the new (default) precedence that follows the usual rules of algebra: the precedence of '`*`' is the same as the precedence of '`/`', so that '`1/2*3`' will equal '`3/2`'.

`-r`
`--round` When converting to a combination of units given by a unit list, round the value of the last unit in the list to the nearest integer.

-S
--show-factor
When converting to a combination of units specified in a list, always show a non-unity factor before a unit that begins with a fraction with a unity denominator. By default, if the unit in a list begins with fraction of the form 1|*x* and its multiplier is an integer other than 1, the fraction is given as the product of the multiplier and the numerator (e.g., '3|8 in' rather than '3 * 1|8 in'). In some cases, this is not what is wanted; for example, the results for a cooking recipe might show '3 * 1|2 cup' as '3|2 cup'. With the --show-factor option, a result equivalent to 1.5 cups will display as '3 * 1|2 cup' rather than '3|2 cup'. A user-specified fractional unit with a numerator other than 1 is never overridden, however—if a unit list specifies '3|4 cup;1|2 cup', a result equivalent to 1 1/2 cups will always be shown as '2 * 3|4 cup' whether or not the --show-factor option is given.

--conformable
In non-interactive mode, show all units conformable with the original unit expression. Only one unit expression is allowed; if you give more than one, units will exit with an error message and return failure.

-v
--verbose
Give slightly more verbose output when converting units. When combined with the -c option this gives the same effect as --check-verbose. When combined with --version produces a more detailed output, equivalent to the --info option.

-V
--version
Print the program version number, tell whether the readline library has been included, tell whether UTF-8 support has been included; give the locale, the location of the default main units data file, and the location of the personal units data file; indicate if the personal units data file does not exist.

When given in combination with the --terse option, the program prints only the version number and exits.

When given in combination with the --verbose option, the program, the --version option has the same effect as the --info option below.

-I
--info
Print the information given with the --version option, show the pathname of the units program, show the status of the UNITSFILE and MYUNITSFILE environment variables, and additional information about how units locates the related files. On systems running Microsoft Windows, the status of the UNITSLOCALE environment variable and information about the related locale map are also given. This option is usually of interest only to developers and administrators, but it can sometimes be useful for troubleshooting.

Combining the --version and --verbose options has the same effect as giving --info.

`-U`
`--unitsfile`
: Print the location of the default main units data file and exit; if the file cannot be found, print "Units data file not found".

`-u` *`units-system`*
`--units` *`units-system`*
: Specify a CGS units system or natural units system. The supported units systems are: gauss[ian], esu, emu, hlu, natural, natural-gauss, hartree, planck, planck-red, and si. See Chapter 8 [Alternative Unit Systems], page 29, for further information about these unit systems.

`-l` *`locale`*
`--locale` *`locale`*
: Force a specified locale such as '`en_GB`' to get British definitions by default. This overrides the locale determined from system settings or environment variables. See Section 16.1 [Locale], page 56, for a description of locale format.

`-n`
`--nolists`
: Disable conversion to unit lists.

`-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.

`-1`
`--one-line`
: Give only one line of output (the forward conversion); do not print the reverse conversion. If a reciprocal conversion is performed, then `units` will still print the "reciprocal conversion" line.

`-t`
`--terse`
: Print only a single conversion factor without any clutter, or if you request a definition, prints just the definition (including its units). This option can be used when calling `units` from another program so that the output is easy to parse. The command `units --terse mile m` produces the output '`1690.344`'. This option has the combined effect of these options: `--strict --quiet --one-line --compact`. When combined with `--version` it produces a display showing only the program name and version number.

`--compact`
: Give compact output featuring only the conversion factor; the multiplication and division signs are not shown, and there is no leading whitespace. If you convert to a unit list, then the output is a semicolon separated list of factors. This turns off the `--verbose` option.

`-q`
`--quiet`
`--silent` Suppress the display of statistics about the number of units loaded, any messages printed by the units database, and the prompting of the user for units. This option does not affect how `units` displays the results. This option is turned on by default if you invoke `units` with a unit expression on the command line.

11 Setting Options Interactively

Many command-line options can also be set interactively, obviating the need to quit and restart `units` to change the values. This can be especially helpful for Windows users who start `units` from a shortcut.

Typing *set* will display a list of all options that can be set interactively, as well as the current and possible values; options set to other than default values have an asterisk ('`*`') prepended. For example,

```
You have: set
  q[uiet] = no          (y|n) do/don't suppress prompting
  o[neline] = no        (y|n) do/don't suppress the second line of output
  st[rict] = no         (y|n) do/don't suppress reciprocal unit conversion
                              (e.g. Hz<->s)
  t[erse] = no          (y|n) do/don't give very terse output
  c[ompact] = no        (y|n) do/don't suppress printing tab, SETFLAG, and '/'
                              characters in results
  v[erbose] = 1         (0|1|2) amount of information shown
 *d[igits] = 9          number of significant digits in output
  e[xponential] = no    (y|n) do/don't use exponential ("scientific") notation
 *f[ormat] = %.9g       printf(3) format specification
  u[nitlists] = yes     (y|n) do/don't allow conversion to unit lists
  r[ound] = no          (y|n) do/don't round last element of unit list output
                              to an integer
  sh[owfactor] = no     (y|n) do/don't show non-unity factor before 1|x
                              in multi-unit output
```

Characters within the square brackets are optional, so settings can be changed by entering only one or two characters.

The syntax for setting options is `set` *option* `=` *value*; the spaces around the '`=`' sign are optional.

Some settings are Boolean, enabled by entering *yes* (or just *y*) and disabled by entering *no* (or just *n*). For example,

```
You have: set quiet = y
  quiet = yes
```

Other settings take an integer value; for example,

```
You have: set d=11
  digits = 11
  format = %.11g
```

The format setting takes a string, the format specification for the printf function in the C programming language; for example,

```
You have: set format = %.9g
  format = %.9g
```

Typing *set option* will display the current value of *option*, for example

```
You have: set u
  unitlists = yes
You have: set d
  digits = 8
  format = %.8g
```

For the digits and exponential options, the value of format is also shown.

12 Scripting with units

Despite its numerous options, units cannot cover every conceivable unit-conversion task. For example, suppose we have found some mysterious scale, but cannot figure out the units in which it is reporting. We reach into our pocket, place a 3.75-gram coin on the scale, and observe the scale reading '0.120'. How do we quickly determine the units? Or we might wonder if a unit has any "synonyms," i.e., other units with the same value.

The capabilities of units are easily extended with simple scripting. Both questions above involve conformable units; on a system with Unix-like utilities, conversions to conformable units could be shown accomplished with the following script:

```
#!/bin/sh

progname=`basename $0 .sh`
umsg="Usage: $progname [<number>] unit"

if [ $# -lt 1 ]
then
    echo "$progname: missing quantity to convert"
    echo "$umsg"
    exit 1
fi

for unit in `units --conformable "$*" | cut -f 1 -d ' '`
do
    echo "$*"   # have -- quantity to convert
    echo $unit  # want -- conformable unit
done | units --terse --verbose
```

When units is invoked with no non-option arguments, it reads *have*/*want* pairs, on alternating lines, from its standard input, so the task can be accomplished with only two invocations of units. This avoids the computational overhead of needlessly reprocessing the units database for each conformable unit, as well as the inherent system overhead of process invocation.

By itself, the script is not very useful. But it could be used in combination with other commands to address specific tasks. For example, running the script through a simple output filter could help solve the scale problem above. If the script is named `conformable`, running

```
$ conformable 3.75g | grep 0.120
```

gives

```
        3.75g = 0.1205653 apounce
        3.75g = 0.1205653 fineounce
        3.75g = 0.1205653 ozt
        3.75g = 0.1205653 tradewukiyeh
        3.75g = 0.1205653 troyounce
```

So we might conclude that the scale is calibrated in troy ounces.

We might run

```
$ units --verbose are
        Definition: 100 m^2 = 100 m^2
```

and wonder if '`are`' has any synonyms, value. To find out, we could run

```
$ conformable are | grep "= 1 "
        are = 1 a
        are = 1 are
```

13 Output Styles

The output can be tweaked in various ways using command line options. With no options, the output looks like this

```
$ units
Currency exchange rates from FloatRates (USD base) on 2023-07-08
3612 units, 109 prefixes, 122 nonlinear units

You have: 23ft
You want: m
        * 7.0104
        / 0.14264521
You have: m
You want: ft;in
        3 ft + 3.3700787 in
```

This is arguably a bit cryptic; the `--verbose` option makes clear what the output means:

```
$ units --verbose
Currency exchange rates from FloatRates (USD base) on 2023-07-08
3612 units, 109 prefixes, 122 nonlinear units

You have: 23 ft
You want: m
        23 ft = 7.0104 m
        23 ft = (1 / 0.14264521) m
You have: meter
You want: ft;in
        meter = 3 ft + 3.3700787 in
```

The `--quiet` option suppresses the clutter displayed when `units` starts, as well as the prompts to the user. This option is enabled by default when you give units on the command line.

```
$ units --quiet
23 ft
m
        * 7.0104
        / 0.14264521

$ units 23ft m
        * 7.0104
        / 0.14264521
```

The remaining style options allow you to display only numerical values without the tab or the multiplication and division signs, or to display just a single line showing the forward conversion:

```
$ units --compact 23ft m
7.0104
0.14264521

$ units --compact m 'ft;in'
3;3.3700787

$ units --one-line 23ft m
        * 7.0104

$ units --one-line 23ft 1/m
        reciprocal conversion
        * 0.14264521

$ units --one-line 23ft kg
conformability error
        7.0104 m
        1 kg
```

Note that when converting to a unit list, the `--compact` option displays a semicolon separated list of results. Also be aware that the `one-line` option doesn't live up to its name if you execute a reciprocal conversion or if you get a conformability error. The former case can be prevented using the `--strict` option, which suppresses reciprocal conversions. Similarly you can suppress unit list conversion using `--nolists`. It is impossible to prevent the three line error output.

```
$ units --compact --nolists m 'ft;in'
Error in 'ft;in': Parse error

$ units --one-line --strict 23ft 1/m
```

The various style options can be combined appropriately. The ultimate combination is the `--terse` option, which combines `--strict`, `--quiet`, `--one-line`, and `--compact` to produce the minimal output, just a single number for regular conversions and a semicolon separated list for conversion to unit lists. This will likely be the best choice for programs that want to call `units` and then process its result.

```
$ units --terse 23ft m
7.0104

$ units --terse m 'ft;in'
3;3.3700787

$ units --terse 23ft 1/m
conformability error
7.0104 m
1 / m

$ units --terse '1 mile'
1609.344 m

$ units --terse mile
5280 ft = 1609.344 m
```

14 Adding Your Own Definitions

14.1 Units Data Files

The units and prefixes that `units` can convert are defined in the units data file, typically `/usr/share/units/definitions.units`. If you can't find this file, run `units --version` to get information on the file locations for your installation. Although you can extend or modify this data file if you have appropriate user privileges, it's usually better to put extensions in separate files so that the definitions will be preserved if you update `units`.

You can include additional data files in the units database using the '`!include`' command in the standard units data file. For example

```
!include    /usr/local/share/units/local.units
```

might be appropriate for a site-wide supplemental data file. The location of the '`!include`' statement in the standard units data file is important; later definitions replace earlier ones, so any definitions in an included file will override definitions before the '!include' statement in the standard units data file. With normal invocation, no warning is given about redefinitions; to ensure that you don't have an unintended redefinition, run `units -c` after making changes to any units data file.

If you want to add your own units in addition to or in place of standard or site-wide supplemental units data files, you can include them in the `.units` file in your home directory. If this file exists it is read after the standard units data file, so that any definitions in this file will replace definitions of the same units in the standard data file or in files included from the standard data file. This file will not be read if any units files are specified on the command line. (Under Windows the personal units file is named `unitdef.units`.) Running `units -V` will display the location and name of your personal units file.

The `units` program first tries to determine your home directory from the `HOME` environment variable. On systems running Microsoft Windows, if `HOME` does not exist, `units` attempts to find your home directory from `HOMEDRIVE`, `HOMEPATH` and `USERPROFILE`. You can specify an arbitrary file as your personal units data file with the `MYUNITSFILE` environment variable; if this variable exists, its value is used without searching your home directory. The default units data files are described in more detail in Chapter 18 [Data Files], page 59.

14.2 Defining New Units and Prefixes

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 '!' must be the first character on the line. The file will be sought in the same directory as the parent file unless you give a full path. The name of the file to be included cannot contain spaces or the comment character '#'.

Unit names cannot begin or end with an underscore ('_') or a decimal point ('.'), and cannot contain any of the operator characters '+', '-', '*', '/', '|', '^', ';', '~', a comma (','), the comment character '#', or parentheses. To facilitate copying and pasting from documents, several typographical characters are converted to operators: the figure dash (U+2012), minus ('−'; U+2212), and en dash ('–'; U+2013) are converted to the operator '-'; the multiplication sign ('×'; U+00D7), N-ary times operator (U+2A09), dot operator ('⋅'; U+22C5), and middle dot ('·'; U+00B7) are converted to the operator '*'; the division sign ('÷'; U+00F7) is converted to the operator '/'; and the fraction slash (U+2044) is converted to the operator '|'; accordingly, none of these characters can appear in unit names.

Names cannot begin with a digit, and if a name ends in a digit other than zero or one, then the name must end with an underscore followed by a string consisting only of digits and decimal points. For example, 'foo_2', 'foo_3.14', and 'foo_3...9' are valid names but 'foo2' or 'foo_a2' are invalid. The underscore is necessary because without it, `units` cannot determine whether 'foo2' is a unit name or represents 'foo^2'. Zero and one are exceptions because `units` never interprets them as exponents.

You could define nitrous oxide as

```
N2O     nitrogen 2  + oxygen
```

but would need to define nitrogen dioxide as

```
NO_2    nitrogen + oxygen 2
```

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 and that there are no circular definitions that lead to endless loops. Because some errors may hide other errors, you should run `units` with the `-c` option again after correcting any errors, and keep doing so until no errors are displayed.

If you define any units that contain '`+`' characters in their definitions, 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 with care, recalling that it has two possible precedence values and may require parentheses to ensure consistent behavior. To compute the difference of '`foo`' and '`bar`' write '`foo+(-bar)`' or even '`foo+-bar`'.

You may wish to intentionally redefine a unit. When you do this, and use the `-c` option, `units` displays a warning message about the redefinition. You can suppress these warnings by redefining a unit using a '`+`' at the beginning of the unit name. Do not include any white space between the '`+`' and the redefined unit name.

Here is an example of a short data 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    72 m             # Inch defined incorrectly terms of meters
ft      12 inches        # The foot defined in terms of inches
mile    5280 ft          # And the mile
+inch   0.0254 m         # Correct redefinition, warning suppressed
```

A unit that 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)`'.

14.3 Defining Nonlinear Units

Some unit conversions of interest are nonlinear; for example, temperature conversions between the Fahrenheit and Celsius scales cannot be done by simply multiplying by conversion 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 that convert to and from linear units in the database, so that an eventual conversion to primitive units is possible.

Here is an example nonlinear unit definition:

```
tempF(x) units=[1;K] domain=[-459.67,) range=[0,) \
            (x+(-32)) degF + stdtemp ; (tempF+(-stdtemp))/degF + 32
```

A nonlinear unit definition comprises a unit name, a formal parameter name, two functions, and optional specifications for units, the domain, and the range (the domain of the inverse function). The functions tell `units` how to convert to and from the new unit. To produce valid results, the arguments of these functions need to have the correct dimensions and be within the domains for which the functions are defined.

The definition begins with the unit name followed immediately (with no spaces) by a '`(`' character. In the parentheses is the name of the formal parameter. Next is an optional specification of the units required by the functions in the definition. In the example above, the '`units=[1;K]`' specification indicates that the '`tempF`' function requires an input argument conformable with '`1`' (i.e., the argument is dimensionless), and that the inverse function requires an input argument conformable with '`K`'. For normal nonlinear units definition, the forward function will always take a dimensionless argument; in general, the inverse function will need units that match the quantity measured by your nonlinear unit. Specifying the units enables `units` to perform error checking on function arguments, and also to assign units to domain and range specifications, which are described later.

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.

To enable conversions to Fahrenheit, 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 and the inverse definition, but then conversions *to* the unit will not be possible. If the inverse definition is omitted, 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 prints an error if it is not valid there, but this is not a guarantee that your inverse is correct.

With some definitions, the units may vary. For example, the definition

```
square(x)       x^2
```

can have any arbitrary units, and can also take dimensionless arguments. In such a case, you should *not* specify units. If a definition takes a root of its arguments, the definition is valid only for units that yield such a root. For example,

```
squirt(x)       sqrt(x)
```

is valid for a dimensionless argument, and for arguments with even powers of units.

Some definitions may not be valid for all real numbers. In such cases, `units` can handle errors better if you specify an appropriate domain and range. You specify the domain and range as shown below:

```
baume(d) units=[1;g/cm^3] domain=[0,130.5] range=[1,10] \
         (145/(145-d)) g/cm^3 ; (baume+-g/cm^3) 145 / baume
```

In this example the domain is specified after '`domain=`' with the two numerical endpoints given in brackets. No spaces can appear before the '`=`'. In accord with mathematical convention, square brackets indicate a closed interval (one that includes its endpoints), and parentheses indicate an open interval (one that does not include its endpoints). An interval can be open or closed on one or both ends; an interval that is unbounded on either end is indicated by omitting the limit on that end. For example, a quantity to which decibel (dB) is applied may have any value greater than zero, so the range is indicated by '`(0,)`':

```
decibel(x) units=[1;1] range=(0,) 10^(x/10); 10 log(decibel)
```

If the domain or range is given, the second endpoint must be greater than the first. The intervals given for the domain and range must be numbers and cannot include units or any calculations.

The domain and range specifications can appear independently and in any order along with the units specification. The values for the domain and range endpoints are attached to the units given in the units specification, and if necessary, the parameter value is adjusted for comparison with the endpoints. For example, if a definition includes '`units=[1;ft]`' and '`range=[3,)`', the range will be taken as 3 ft to infinity. If the function is passed a parameter of '`900 mm`', that value will be adjusted to 2.9527559 ft, which is outside the specified range. If you omit the units specification from the previous example, `units` can not tell whether you intend the lower endpoint to be 3 ft or 3 microfurlongs, and can not adjust the parameter value of 900 mm for comparison. Without units, numerical values other than zero or plus or minus infinity for domain or range endpoints are meaningless, and accordingly they are not allowed. If you give other values without units, then the definition will be ignored and you will get an error message.

Although the units, domain, and range specifications are optional, it's best to give them when they are applicable; doing so allows `units` to perform better error checking and give more helpful error messages. Giving the domain and range also enables the `--check` option to find a point in the domain to use for its point check of your inverse definition.

You can make synonyms for nonlinear units by providing 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) units=[1;K] tempF(x); ~tempF(fahrenheit)
```

This is useful for creating a nonlinear unit definition that differs slightly from an existing definition without having to repeat the original functions. For example,

```
dBW(x)     units=[1;W] range=[0,) dB(x) W ;  ~dB(dBW/W)
```

If you wish a synonym to refer to an existing nonlinear unit without modification, you can do so more simply by adding the synonym with appended parentheses as a new unit, with the existing nonlinear unit—without parentheses—as the definition. So to create a synonym for 'tempF' you could write

```
fahrenheit()  tempF
```

The definition must be a nonlinear unit; for example, the synonym

```
fahrenheit()  meter
```

will result in an error message when `units` starts.

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. This definition requires a length as input and produces an area as output, as indicated by the 'units=' specification. Specifying the range as the nonnegative numbers can prevent cryptic error messages.

```
circlearea(r) units=[m;m^2] range=[0,)   pi r^2 ; sqrt(circlearea/pi)
```

14.4 Defining Piecewise Linear Units

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 invalid; 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 preceding 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.

After adding nonlinear units definitions, you should normally run '`units --check`' to check for errors. If the '`units`' keyword is not given, the `--check` option checks a nonlinear unit definition using a dimensionless argument, and then checks using an arbitrary combination of units, as well as the square and cube of that combination; a warning is given if any of these tests fail. For example,

```
Warning: function 'squirt(x)' defined as 'sqrt(x)'
         failed for some test inputs:
         squirt(7(kg K)^1): Unit not a root
         squirt(7(kg K)^3): Unit not a root
```

Running '`units --check`' will print a warning if a non-monotonic piecewise linear unit is encountered. For example, the relationship between ANSI coated abrasive designation and mean particle size is non-monotonic in the vicinity of 800 grit:

```
ansicoated[micron] \
     . . .
    600 10.55 \
    800 11.5 \
    1000 9.5 \
```

Running '`units --check`' would give the error message

```
Table 'ansicoated' lacks unique inverse around entry 800
```

Although the inverse is not well defined in this region, it's not really an error. Viewing such error messages can be tedious, and if there are enough of them, they can distract from true errors. Error checking for nonlinear unit definitions can be suppressed by giving the '`noerror`' keyword; for the examples above, this could be done as

```
squirt(x) noerror domain=[0,) range=[0,) sqrt(x); squirt^2
ansicoated[micron] noerror \
     . . .
```

Use the '`noerror`' keyword with caution. The safest approach after adding a nonlinear unit definition is to run '`units --check`' and confirm that there are no actual errors before adding the '`noerror`' keyword.

14.5 Defining Multivariate Functions

Some common dimensional calculations such as wind chill or body mass index have more than one parameter. To support these calculations, `units` allows multivariate functions as unit definitions. Unlike the nonlinear units described above, multivariate functions provide only a forward evaluation and are not invertible: it is impossible to convert a wind chill output back to the temperature and wind speed that generated it.

A multivariate function definition is similar to that for other nonlinear units. For example, you could define '`bmi`' like this, starting with the unit's name followed immediately without white space by a comma-separated parameter list:

```
bmi(ht,wt) units=[m,kg] domain=(0,)(0,) (wt/kg)/(ht/m)^2
```

This defines a new unit, 'bmi', that has two parameters, 'height', and 'weight'. You are allowed to use white space in the parameter list. The optional 'units=' specification (with no space before the '=') provides a comma-separated list in square brackets of units that provide the dimensions of the parameters. (Since multivariate units require the use of commas, you cannot use them to specify the units in the 'units=' specification.) If you list fewer units than the number of parameters, the remaining parameters will allow any unit. For univariate nonlinear units, a semicolon introduces the units of the inverse. The semicolon is not permitted in the 'units=' list for multivariate functions.

You specify the domain, if desired, using 'domain=', followed (without white space) by the intervals defining the domain of each parameter, either juxtaposed or separated by commas. The above example requires that both arguments be strictly positive. As usual, use square brackets to indicate closed intervals. The rules for domain intervals are the same as those described above for nonlinear units. If you need to indicate an unconstrained input give the interval '(,)'. The intervals in the domain specification must use numerical values; you cannot use units or calculations in the domain.

The final part of the definition is the expression that defines the unit. Since inverses are not well-defined for multivariate units, you cannot provide an inverse. Since you cannot provide an inverse, you also cannot provide a range, so the 'range=' specification is invalid for multivariate functions.

If you need to specify that a particular unit is of arbitrary dimension when units appearing after it in the parameter list are of specified dimension, you can do this using the special dimension specification, '*'. Here is a toy example:

```
func(a,b) units=[*,1] domain=(,),[-1,1] a asin(b)
```

In this example the first argument is permitted to be of any dimension, but the second argument must be dimensionless. The first argument has no domain bounds, but the second one must lie in the domain of the arcsine function.

14.6 Defining Unit List Aliases

Unit list aliases are treated differently from unit definitions, because they are a data entry shorthand rather than a true definition for a new unit. A unit list alias definition begins with '!unitlist' and includes the alias and the definition; for example, the aliases included in the standard units data file are

```
!unitlist   hms     hr;min;sec
!unitlist   time    year;day;hr;min;sec
!unitlist   dms     deg;arcmin;arcsec
!unitlist   ftin    ft;in;1|8 in
!unitlist   usvol   cup;3|4 cup;2|3 cup;1|2 cup;1|3 cup;1|4 cup;\
                    tbsp;tsp;1|2 tsp;1|4 tsp;1|8 tsp
```

Unit list aliases are only for unit lists, so the definition must include a ';'. Unit list aliases can never be combined with units or other unit list aliases, so the definition of 'time' shown above could *not* have been shortened to 'year;day;hms'.

As usual, be sure to run 'units --check' to ensure that the units listed in unit list aliases are conformable.

15 Numeric Output Format

By default, `units` shows results to eight significant digits in general number format. You can change this with the `--exponential`, `--digits`, and `--output-format` options. The first sets an exponential format (i.e., scientific notation) like that used in the original Unix `units` program, the second allows you to specify a different number of significant digits, and the last allows you to control the output appearance using the format for the `printf` function in the C programming language. If you only want to change the number of significant digits or specify exponential format type, use the `--digits` and `--exponential` options. The `--output-format` option affords the greatest control of the output appearance, but requires at least rudimentary knowledge of the `printf` format syntax. See Chapter 10 [Invoking Units], page 35, for descriptions of these options.

15.1 Format Specification

The format specification recognized with the `--output-format` option is a subset of that for `printf`. The format specification has the form `%`[*flags*][*width*][`.`*precision*]*type*; it must begin with '`%`', and must end with a floating-point type specifier: '`g`' or '`G`' to specify the number of significant digits, '`e`' or '`E`' for scientific notation, and '`f`' for fixed-point decimal. The ISO C99 standard added the '`F`' type for fixed-point decimal and the '`a`' and '`A`' types for hexadecimal floating point; these types are allowed with compilers that support them. Type length modifiers (e.g., '`L`' to indicate a long double) are inapplicable and are not allowed.

The default format for `units` is '`%.8g`'; for greater precision, you could specify `-o %.15g`. The '`g`' and '`G`' format types use exponential format whenever the exponent would be less than -4, so the value 0.000013 displays as '`1.3e-005`'. These types also use exponential notation when the exponent is greater than or equal to the precision, so with the default format, the value 5×10^7 displays as '`50000000`' and the value 5×10^8 displays as '`5e+008`'. If you prefer fixed-point display, you might specify `-o %.8f`; however, small numbers will display very few significant digits, and values less than 5×10^{-8} will show nothing but zeros.

The format specification may include one or more optional flags: '`+`', '` `' (space), '`#`', '`-`', or '`0`' (the digit zero). The digit-grouping flag '`'`' (apostrophe) is allowed with compilers that support it. Flags are followed by an optional value for the minimum field width, and an optional precision specification that begins with a period (e.g., '`.6`'). The field width includes the digits, decimal point, the exponent, thousands separators (with the digit-grouping flag), and the sign if any of these are shown.

15.2 Flags

The '`+`' flag causes the output to have a sign ('`+`' or '`-`'). The space flag '` `' is similar to the '`+`' flag, except that when the value is positive, it is prefixed with a space rather than a plus sign; this flag is ignored if the '`+`' flag is also given. The '`+`' or '` `' flag could be useful if conversions might include positive and negative results, and you wanted to align the decimal points in exponential notation. The '`#`' flag causes the output value to contain a decimal point in all cases; by default, the output contains a decimal point only if there are digits (which can be trailing zeros) to the right of the point. With the '`g`' or '`G`' types, the '`#`' flag also prevents the suppression of trailing zeros. The digit-grouping flag '`'`' shows a thousands separator in digits to the left of the decimal point. This can be useful when displaying large numbers in fixed-point decimal; for example, with the format '%f',

```
You have: mile
You want: microfurlong
        * 8000000.000000
        / 0.000000
```

the magnitude of the first result may not be immediately obvious without counting the digits to the left of the decimal point. If the thousands separator is the comma (','), the output with the format '%'f' might be

```
You have: mile
You want: microfurlong
        * 8,000,000.000000
        / 0.000000
```

making the magnitude readily apparent. Unfortunately, few compilers support the digit-grouping flag.

With the '-' flag, the output value is left aligned within the specified field width. If a field width greater than needed to show the output value is specified, the '0' (zero) flag causes the output value to be left padded with zeros until the specified field width is reached; for example, with the format '%011.6f',

```
You have: troypound
You want: grain
        * 5760.000000
        / 0000.000174
```

The '0' flag has no effect if the '-' (left align) flag is given.

15.3 Field Width

By default, the output value is left aligned and shown with the minimum width necessary for the specified (or default) precision. If a field width greater than this is specified, the value shown is right aligned, and padded on the left with enough spaces to provide the specified field width. A width specification is typically used with fixed-point decimal to have columns of numbers align at the decimal point; this arguably is less useful with `units` than with long columnar output, but it may nonetheless assist in quickly assessing the relative magnitudes of results. For example, with the format '%12.6f',

```
You have: km
You want: in
        * 39370.078740
        /     0.000025
You have: km
You want: rod
        *   198.838782
        /     0.005029
You have: km
You want: furlong
        *     4.970970
        /     0.201168
```

15.4 Precision

The meaning of "precision" depends on the format type. With 'g' or 'G', it specifies the number of significant digits (like the --digits option); with 'e', 'E', 'f', or 'F', it specifies the maximum number of digits to be shown after the decimal point.

With the 'g' and 'G' format types, trailing zeros are suppressed, so the results may sometimes have fewer digits than the specified precision (as indicated above, the '#' flag causes trailing zeros to be displayed).

The default precision is 6, so '%g' is equivalent to '%.6g', and would show the output to six significant digits. Similarly, '%e' or '%f' would show the output with six digits after the decimal point.

The C printf function allows a precision of arbitrary size, whether or not all of the digits are meaningful. With most compilers, the maximum internal precision with units is 15 decimal digits (or 13 hexadecimal digits). With the --digits option, you are limited to the maximum internal precision; with the --output-format option, you may specify a precision greater than this, but it may not be meaningful. In some cases, specifying excess precision can result in rounding artifacts. For example, a pound is exactly 7000 grains, but with the format '%.18g', the output might be

```
You have: pound
You want: grain
        * 6999.9999999999991
        / 0.00014285714285714287
```

With the format '%.25g' you might get the following:

```
You have: 1/3
You want:
        Definition: 0.333333333333333314829616256247
```

In this case the displayed value includes a series of digits that represent the underlying binary floating-point approximation to 1/3 but are not meaningful for the desired computation. In general, the result with excess precision is system dependent. The precision affects only the *display* of numbers; if a result relies on physical constants that are not known to the specified precision, the number of physically meaningful digits may be less than the number of digits shown.

See the documentation for printf for more detailed descriptions of the format specification.

The --output-format option is incompatible with the --exponential or --digits options; if the former is given in combination with either of the latter, the format is controlled by the last option given.

16 Localization

Some units have different values in different locations. The localization feature accommodates this by allowing a units data file to specify definitions that depend on the user's locale.

16.1 Locale

A locale is a subset of a user's environment that indicates the user's language and country, and some attendant preferences, such as the formatting of dates. The `units` program attempts to determine the locale from the POSIX `setlocale` function; if this cannot be done, `units` examines the environment variables `LC_CTYPE` and `LANG`. On POSIX systems, a locale is of the form *language_country*, where *language* is the two-character code from ISO 639-1 and *country* is the two-character code from ISO 3166-1; *language* is lower case and *country* is upper case. For example, the POSIX locale for the United Kingdom is `en_GB`.

On systems running Microsoft Windows, the value returned by `setlocale` is different from that on POSIX systems; `units` attempts to map the Windows value to a POSIX value by means of a table in the file `locale_map.txt` in the same directory as the other data files. The file includes entries for many combinations of language and country, and can be extended to include other combinations. The `locale_map.txt` file comprises two tab-separated columns; each entry is of the form

Windows-locale *POSIX-locale*

where *POSIX-locale* is as described above, and *Windows-locale* typically spells out both the language and country. For example, the entry for the United States is

```
English_United States   en_US
```

You can force `units` to run in a desired locale by using the `-l` option.

In order to create unit definitions for a particular locale you begin a block of definitions in a unit datafile with '`!locale`' followed by a locale name. The '`!`' must be the first character on the line. The `units` program reads the following definitions only if the current locale matches. You end the block of localized units with '`!endlocale`'. Here is an example, which defines the British gallon.

```
!locale en_GB
gallon       4.54609 liter
!endlocale
```

16.2 Additional Localization

Sometimes the locale isn't sufficient to determine unit preferences. There could be regional preferences, or a company could have specific preferences. Though probably uncommon, such differences could arise with the choice of English customary units outside of English-speaking countries. To address this, `units` allows specifying definitions that depend on environment variable settings. The environment variables can be controlled based on the current locale, or the user can set them to force a particular group of definitions.

A conditional block of definitions in a units data file begins with either '`!var`' or '`!varnot`' following by an environment variable name and then a space separated list of values. The leading '`!`' must appear in the first column of a units data file, and the conditional block is terminated by '`!endvar`'. Definitions in blocks beginning with '`!var`' are executed only if the environment variable is exactly equal to one of the listed values. Definitions in blocks beginning with '`!varnot`' are executed only if the environment variable does *not* equal any of the list values.

The inch has long been a customary measure of length in many places. The word comes from the Latin *uncia* meaning "one twelfth," referring to its relationship with the foot. By the 20th century, the inch was officially defined in English-speaking countries relative to the yard, but until 1959, the yard differed slightly among those countries. In France the customary inch, which was displaced in 1799 by the meter, had a different length based on a french foot. These customary definitions could be accommodated as follows:

```
!var INCH_UNIT usa
yard          3600|3937 m
!endvar
!var INCH_UNIT canada
yard          0.9144 meter
!endvar
!var INCH_UNIT uk
yard          0.91439841 meter
!endvar
!var INCH_UNIT canada uk usa
foot          1|3 yard
inch          1|12 foot
!endvar
!var INCH_UNIT france
foot          144|443.296 m
inch          1|12 foot
line          1|12 inch
!endvar
!varnot INCH_UNIT usa uk france canada
!message Unknown value for INCH_UNIT
!endvar
```

When `units` reads the above definitions it will check the environment variable `INCH_UNIT` and load only the definitions for the appropriate section. If `INCH_UNIT` is unset or is not set to one of the four values listed, then `units` will run the last block. In this case that block uses the '`!message`' command to display a warning message. Alternatively that block could set default values.

In order to create default values that are overridden by user settings the data file can use the '`!set`' command, which sets an environment variable *only if it is not already set*; these settings are only for the current `units` invocation and do not persist. So if the example above were preceded by '`!set INCH_UNIT france`', then this would make '`france`' the default value for `INCH_UNIT`. If the user had set the variable in the environment before invoking `units`, then `units` would use the user's value.

To link these settings to the user's locale you combine the '`!set`' command with the '`!locale`' command. If you wanted to combine the above example with suitable locales you could do by *preceding* the above definition with the following:

```
!locale en_US
!set INCH_UNIT usa
!endlocale
!locale en_GB
!set INCH_UNIT uk
!endlocale
!locale en_CA
!set INCH_UNIT canada
!endlocale
!locale fr_FR
!set INCH_UNIT france
!endlocale
!set INCH_UNIT france
```

These definitions set the overall default for `INCH_UNIT` to '`france`' and set default values for four locales appropriately. The overall default setting comes last so that it only applies when `INCH_UNIT` was not set by one of the other commands or by the user.

If the variable given after '`!var`' or '`!varnot`' is undefined, then `units` prints an error message and ignores the definitions that follow. Use '`!set`' to create defaults to prevent this situation from arising. The `-c` option only checks the definitions that are active for the current environment and locale, so when adding new definitions take care to check that all cases give rise to a well defined set of definitions.

17 Environment Variables

The `units` program uses the following environment variables:

`HOME`
: Specifies the location of your home directory; it is used by `units` to find a personal units data file '`.units`'. On systems running Microsoft Windows, the file is '`unitdef.units`', and if `HOME` does not exist, `units` tries to determine your home directory from the `HOMEDRIVE` and `HOMEPATH` environment variables; if these variables do not exist, units finally tries `USERPROFILE`—typically `C:\Users\`*username* (Windows Vista and Windows 7) or `C:\Documents and Settings\`*username* (Windows XP).

`LC_CTYPE, LANG`
: Checked to determine the locale if `units` cannot obtain it from the operating system. Sections of the default main units data file are specific to certain locales.

`MYUNITSFILE`
: Specifies your personal units data file. If this variable exists, `units` uses its value rather than searching your home directory for '`.units`'. The personal units file will not be loaded if any data files are given using the `-f` option.

`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 '`+n`'*n* syntax for specifying a line number. The default pager is `more`; `PAGER` can be used to specify alternatives such as `less`, `pg`, `emacs`, or `vi`.

UNITS_ENGLISH
: Set to either 'US' or 'GB' to choose United States or British volume definitions, overriding the default from your locale.

UNITSFILE
: Specifies the units data file to use (instead of the default). You can only specify a single units data file using this environment variable. If units data files are given using the -f option, the file specified by UNITSFILE will be not be loaded unless the -f option is given with the empty string ('units -f ""').

UNITSLOCALEMAP
: Windows only; this variable has no effect on Unix-like systems. Specifies the units locale map file to use (instead of the default). This variable seldom needs to be set, but you can use it to ensure that the locale map file will be found if you specify a location for the units data file using either the -f option or the UNITSFILE environment variable, and that location does not also contain the locale map file.

UNITS_SYSTEM
: This environment variable is used in the default main data file to select CGS measurement systems. Currently supported systems are 'esu', 'emu', 'gauss[ian]', 'hlu', 'natural', 'natural-gauss', 'planck', 'planck-red', 'hartree' and 'si'. The default is 'si'.

18 Data Files

The units program uses four default data files: the main data file, definitions.units; the atomic masses of the elements, elements.units; currency exchange rates, currency.units, and the US Consumer Price Index, cpi.units. The last three files are loaded by means of '!include' directives in the main file (see Chapter 22 [Database Command Syntax], page 64). The program can also use an optional personal units data file .units (unitdef.units under Windows) located in the user's home directory. The personal units data file is described in more detail in Section 14.1 [Units Data Files], page 45.

On Unix-like systems, the data files are typically located in /usr/share/units if units is provided with the operating system, or in /usr/local/share/units if units is compiled from the source distribution. Note that the currency file currency.units is a symbolic link to another location.

On systems running Microsoft Windows, the files may be in the same locations if Unix-like commands are available, a Unix-like file structure is present (e.g., C:/usr/local), and units is compiled from the source distribution. If Unix-like commands are not available, a more common location is C:\Program Files (x86)\GNU\units (for 64-bit Windows installations) or C:\Program Files\GNU\units (for 32-bit installations).

If units is obtained from the GNU Win32 Project (http://gnuwin32.sourceforge.net/), the files are commonly in C:\Program Files\GnuWin32\share\units.

If the default main units data file is not an absolute pathname, units will look for the file in the directory that contains the units program; if the file is not found there, units will look in a directory ../share/units relative to the directory with the units program.

You can determine the location of the files by running '`units --version`'. Running '`units --info`' will give you additional information about the files, how `units` will attempt to find them, and the status of the related environment variables.

19 Unicode Support

The standard units data file is in Unicode, using UTF-8 encoding. Most definitions use only ASCII characters (i.e., code points U+0000 through U+007F); definitions using non-ASCII characters appear in blocks beginning with '`!utf8`' and ending with '`!endutf8`'.

The non-ASCII definitions are loaded only if the platform and the locale support UTF-8. Platform support is determined when `units` is compiled; the locale is checked at every invocation of `units`. To see if your version of `units` includes Unicode support, invoke the program with the `--version` option.

When Unicode support is available, `units` checks every line within UTF-8 blocks in all of the units data files for invalid or non-printing UTF-8 sequences; if such sequences occur, `units` ignores the entire line. In addition to checking validity, `units` determines the display width of non-ASCII characters to ensure proper positioning of the pointer in some error messages and to align columns for the '`search`' and '`?`' commands.

Microsoft Windows supports UTF-8 in console applications running in Windows Terminal; UTF-8 is not supported in applications running in the older Windows Console Host—see Section 19.1 [Unicode Support on Windows], page 61. The UTF-16 and UTF-32 encodings are not supported on any platforms.

If Unicode support is available and definitions that contain non-ASCII UTF-8 characters are added to a units data file, those definitions should be enclosed within '`!utf8`' ... '`!endutf8`' to ensure that they are only loaded when Unicode support is available. As usual, the '`!`' must appear as the first character on the line. As discussed in Section 14.1 [Units Data Files], page 45, it's usually best to put such definitions in supplemental data files linked by an '`!include`' command or in a personal units data file.

When Unicode support is not available, `units` makes no assumptions about character encoding, except that characters in the range 00–7F hexadecimal correspond to ASCII encoding. Non-ASCII characters are simply sequences of bytes, and have no special meanings; for definitions in supplementary units data files, you can use any encoding consistent with this assumption. For example, if you wish to use non-ASCII characters in definitions when running `units` under Windows, you can use a character set such as Windows "ANSI" (code page 1252 in the US and Western Europe); if this is done, the console code page must be set to the same encoding for the characters to display properly. You can even use UTF-8, though some messages may be improperly aligned, and `units` will not detect invalid UTF-8 sequences. If you use UTF-8 encoding when Unicode support is not available, you should place any definitions with non-ASCII characters *outside* '`!utf8`' ... '`!endutf8`' blocks—otherwise, they will be ignored.

Except for code examples, typeset material usually uses the Unicode symbols for mathematical operators. To facilitate copying and pasting from such sources, several typographical characters are converted to the ASCII operators used in `units`: the figure dash (U+2012), minus ('−'; U+2212), and en dash ('–'; U+2013) are converted to the operator '`-`'; the multiplication sign ('×'; U+00D7), N-ary times operator (U+2A09), dot operator ('⋅'; U+22C5), and middle dot ('·'; U+00B7) are converted to the operator '*'; the division sign ('÷'; U+00F7) is converted to the operator '/'; and the fraction slash (U+2044) is converted to the operator '|'.

19.1 Unicode Support on Windows

Microsoft Windows supports UTF-8 in console applications running in Windows Terminal but not in applications running in the older Windows Console Host. In Windows Terminal, the code page must be set to 65001 for UTF-8 to be enabled. With the UTF-8 code page, running `units -V` might show

```
GNU Units version 2.24
Without readline, with UTF-8, locale English_United States (en_US)
```

Two values are shown for the locale: the first is the one returned by the system; the second is the POSIX value to which the system value is mapped.

With a different code page, the result might be

```
GNU Units version 2.24
Without readline, with UTF-8 (disabled), locale English_United States (en_US)
To enable UTF-8: set code page to 65001
```

If `units` is running in Windows Console Host, regardless of the code page, the result might be

```
GNU Units version 2.24
Without readline, with UTF-8 (disabled), locale English_United States (en_US)
To enable UTF-8: run in Windows Terminal and set code page to 65001
```

The UTF-8 code page can be set by running `chcp 65001`.

As of late 2024, the Windows build of `units` does not identify characters—typically East Asian—that occupy more than one column, and error messages involving those characters may not be properly aligned.

20 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 `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 `?`, then `units` will display a list of all the units that 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 unit name, 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.

The readline library also keeps a history of the values you enter. You can move through this history using the up and down arrows. The history is saved to the file .units_history in your home directory so that it will persist across multiple units invocations. If you wish to keep work for a certain project separate you can change the history filename using the --history option. You could, for example, make an alias for units to units --history .units_history so that units would save separate history in the current directory. The length of each history file is limited to 5000 lines. Note also that if you run several concurrent copies of units each one will save its new history to the history file upon exit.

21 Updating Currency Exchange Rates and CPI

21.1 Currency Exchange Rates

The units program database includes currency exchange rates and prices for some precious metals. Of course, these values change over time, sometimes very rapidly, and units cannot provide real-time values. To update the exchange rates, run units_cur, which rewrites the file containing the currency rates, typically /var/lib/units/currency.units or /usr/local/com/units/currency.units on a Unix-like system or C:\Program Files (x86)\GNU\units\definitions.units on a Windows system.

This program requires Python 3 (https://www.python.org). The program must be run with suitable permissions to write the file. To keep the rates updated automatically, run it using a cron job on a Unix-like system, or a similar scheduling program on a different system.

Reliable free sources of currency exchange rates have been annoyingly ephemeral. The program currently supports several sources:

- ExchangeRate-API.com (https://www.exchangerate-api.com).
  The default currency server. Allows open access without an API key, with unlimited API requests. Rates update once a day, the US dollar ('USD') is the default base currency, and you can choose your base currency with the -b option described below. You can optionally sign up for an API key to access paid benefits such as faster data update rates.
- FloatRates (https://www/floatrates.com).
  The US dollar ('USD') is the default base currency. You can change the base currency with the -b option described below. Allowable base currencies are listed on the FloatRates website. Exchange rates update daily.
- The European Central Bank (https://www.ecb.europa.eu).
  The base currency is always the euro ('EUR'). Exchange rates update daily. This source offers a more limited list of currencies than the others.
- Fixer (https://fixer.io).
  Registration for a free API key is required. With a free API key, base currency is the euro; exchange rates are updated hourly, the service has a limit of 1,000 API calls per

month, and SSL encryption (https protocol) is not available. Most of these restrictions are eliminated or reduced with paid plans.

- open exchange rates (`https://openexchangerates.org`).
  Registration for a free API key is required. With a free API key, the base currency is the US dollar; exchange rates are updated hourly, and there is a limit of 1,000 API calls per month. Most of these restrictions are eliminated or reduced with paid plans.

The default source is FloatRates; you can select a different one using `-s` option described below.

Precious metals pricing is obtained from Packetizer (`www.packetizer.com`). This site updates once per day.

21.2 US Consumer Price Index

The `units` program includes the US Consumer Price Index (CPI) published by the US Bureau of Labor Statistics: specifically, the Consumer Price Index for All Urban Consumers (CPI-U), not seasonally adjusted—Series CUUR0000SA0. The `units_cur` command updates the CPI and saves the result in `cpi.units` in the same location as `currency.units`. The data are obtained via the BLS Public Data API (`https://www.bls.gov/developers/`). These data update once a month. When `units_cur` runs it will only attempt to update the CPI data if the current CPI data file is from a previous month, or if the current date is after the 18th of the month.

21.3 Invoking `units_cur`

You invoke `units_cur` like this:

```
units_cur [options] [currency_file] [cpi_file]
```

By default, the output is written to the default currency and CPI files described above; this is usually what you want, because this is where `units` looks for the files. If you wish, you can specify different filenames on the command line and `units_cur` will write the data to those files. If you give '`-`' for a file it will write to standard output.

The following options are available:

`-h`
`--help` Print a summary of the options for `units_cur`.

`-V`
`--version`
Print the `units_cur` version number.

`-v`
`--verbose`
Give slightly more verbose output when attempting to update currency exchange rates.

`-s` *source*
`--source` *source*
Specify the source for currency exchange rates; currently supported values are '`floatrates`' (for FloatRates), '`eubank`' (for the European Central Bank),

'`fixer`' (for Fixer), and '`openexchangerates`' (for open exchange rates); the last two require an API key to be given with the `-k` option.

`-b` *base*
`--base` *base*
: Set the base currency (when allowed by the site providing the data). *base* should be a 3-letter ISO currency code, e.g., '`USD`'. The specified currency will be the primitive currency unit used by `units`. You may find it convenient to specify your local currency. Conversions may be more accurate and you will be able to convert to your currency by simply hitting `Enter` at the '`You want:`' prompt. This option is ignored if the source does not allow specifying the base currency. (Currently only floatrates supports this option.)

`-k` *key*
`--key` *key*
: Set the API key to *key* for currency sources that require it.

`--blskey` *BLSkey*
: Set the US Bureau of Labor Statistics (BLS) key for fetching CPI data. Without a BLS key you should be able to fetch the CPI data exactly one time per day. If you want to use a key you must request a personal key from BLS.

22 Database Command Syntax

unit definition
: Define a regular unit.

prefix- definition
: Define a prefix.

funcname(*var*) `noerror units=[`*in-units*,*out-units*`] domain=[`*x1*,*x2*`] range=[`*y1*,*y2*`]` *definition(var)* `;` *inverse(funcname)*
: Define a nonlinear unit or unit function. The four optional keywords `noerror`, '`units=`', '`range=`' and '`domain=`' can appear in any order. The definition of the inverse is optional.

tabname[*out-units*] `noerror` *pair-list*
: Define a piecewise linear unit. The pair list gives the points on the table listed in ascending order. The `noerror` keyword is optional.

`!endlocale`
: End a block of definitions beginning with '`!locale`'

`!endutf8`
: End a block of definitions begun with '`!utf8`'

`!endvar`
: End a block of definitions begun with '`!var`' or '`!varnot`'

`!include` *file*
: Include the specified file.

`!locale` *value*
: Load the following definitions only of the locale is set to *value*.

`!message text`
: Display *text* when the database is read unless the quiet option (`-q`) is enabled. If you omit *text*, then units will display a blank line. Messages will also appear in the log file.

`!prompt text`
: Prefix the '`You have:`' prompt with the specified text. If you omit *text*, then any existing prefix is canceled.

`!set variable value`
: Sets the environment variable, *variable*, to the specified value *only if* it is not already set.

`!unitlist alias definition`
: Define a unit list alias.

`!utf8`
: Load the following definitions only if `units` is running with UTF-8 enabled.

`!var envar value-list`
: Load the block of definitions that follows only if the environment variable *envar* is set to one of the values listed in the space-separated value list. If *envar* is not set, `units` prints an error message and ignores the block of definitions.

`!varnot envar value-list`
: Load the block of definitions that follows only if the environment variable *envar* is set to value that is *not* listed in the space-separated value list. If *envar* is not set, `units` prints an error message and ignores the block of definitions.

23 GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

   Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

E

F

H

I

L

M

N

O