Shell Scripting Recipes

A Problem-Solution Approach

CHRIS F. A. JOHNSON

Apress®

Shell Scripting Recipes: A Problem-Solution Approach

ISBN (pbk): 1-59059-471-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jason Gilmore
Technical Reviewer: William Park
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser
Assistant Publisher: Grace Wong
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Scott Carter
Production Manager: Kari Brooks-Copony
Compositor and Artist: Kinetic Publishing Services, LLC
Proofreader: Linda Seifert
Indexer: James Minkin
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 8

The Dating Game

The Y2K bug is all but forgotten, and the Unix 2038 bug should be a non-issue (unless you are still using software that was compiled more than 30 years earlier).[1] Date-calculating software in use today should be good to the year 9999, and perhaps beyond. Unless, that is, adjustments have been made to the calendar to accommodate the couple of days discrepancy that will have crept in by then.

Although the uproar reached a peak in the months before January 1, 2000, the Y2K problem had been felt for many years. There were programs that had to deal with people born before 1900 and others (or even the same people) who would live into the twenty-first century. Two digits was inadequate. Some software worked around the problem by adding 1900 when the two digits exceeded a certain threshold (say, 30), and adding 2000 when it was less. That still didn't work if you were dealing both with people born before 1930 and people who could live until after 2030.

For no good reason, programmers continued to enter and store a year as two digits. There was a quip making the rounds on the Internet in the late 1990s saying, "Trust programmers to shorten the year 2000 to Y2K; that's what caused the problem in the first place." Well, no, it's not. The Y2K bug was caused by dates being stored with only the last two digits of the year. Information was lost by doing that; no information is lost in the abbreviation Y2K.

New Year's Eve 1999 slid into New Year's Day 2000 with hardly a ripple. Was that because the Y2K problem had been overstated, or because a sterling job had been done in fixing all extant software? I think it was a bit of both. I occasionally see dates in this century displayed as 12/12/104, because the year is stored in two digits; and a lot of systems were upgraded unnecessarily because of uncertainty about the bug.

When working on a system that has very little memory or storage space, cramming information into the smallest space possible makes sense. I did it on 8-bit machines, but with today's computers, space is so cheap, and compression so efficient, that there's no excuse for using two-digit years any more (small, embedded systems might be an exception, but even then, there should be better ways of saving space).

---

1. On Unix, the system time is stored as seconds since the epoch (1 January 1970) in a 32-bit integer. The maximum capacity of such an integer will be reached on 19 January 2038. This will be (and already is being) solved by using a 64-bit integer, whether on a 64-bit computer with a native integer of that size, or with the field being defined as a 64-bit integer on a 32-bit computer. For a more complete discussion, see *The Year 2038 Problem* by Roger M. Wilcox at `http://pw2.netcom.com/~rogermw/Y2038.html`.

This chapter gives you more than a dozen functions for manipulating and formatting dates. With them you can convert a date in almost any format to an ISO standard date (e.g., "28 February 2005" or "feb 28 2005" to 2005-02-28). You can calculate how many days late your creditors are, or whether any given date is a valid date. You can find the day of the week on which you were born, or how many days old you are. You can find yesterday's date, tomorrow's date, or the date 666 days from now. You can store consistent ISO dates and quickly display them in a user-friendly format. And you can use these functions to build many more date commands that I didn't have room for.

The date-funcs Library

The scripts in this chapter never use anything but the full year, whether as input or storage. The only shortcut taken is that they are limited to the Gregorian calendar. Since most of the world had switched from the Julian calendar by the end of 1752, there is little practical consequence.

8.1 split_date—Divide a Date into Day, Month, and Year

Dates often arrive in a program in a format such as the International Standard (ISO 8601), 2005-03-01, or the common usage of 3/7/2004 (or the unspeakable 4/5/06). For most programming purposes, this needs to be translated into either three separate variables or a single integer (which requires the three values for its calculation). A script must be able to split the date and assign its components to the correct variables (is 3/7/2004 the 3rd of July or the 7th of March?). The conversion to an integer comes later in the chapter.

How It Works

Using a customized field separator, `IFS`, `split-date` breaks the date into three parts and assigns each part to a variable specified on the command line. Leading zeroes are stripped, so the results can be used in arithmetic expressions.

Usage

```
split_date "DATE" [VAR1 [VAR2 [VAR3]]]
```

The elements of the date may be separated by whitespace, or a hyphen, period or slash. If the date contains whitespace, it must be quoted on the command line:

```
$ format="  Day: %s\nMonth: %s\n Year: %s\n"
$ split_date "May 5 1977" month day year
$ printf "$format" "$day" "$month" "$year"
  Day: 5
Month: May
 Year: 1977
```

If no variables are specified on the command line, defaults are used, assigning the first part to `SD_YEAR`, the second to `SD_MONTH`, and the third to `SD_DAY`:

```
$ split_date "1949-09-29"
$ printf "$format" "$SD_DAY" "$SD_MONTH" "$SD_YEAR"
  Day: 29
Month: 9
 Year: 1949
```

The `split_date` function makes no assumptions about the validity of the components; the only change it makes to its input is that it removes a leading zero, if there is one, so that the resulting number can be used in arithmetic expressions.

The Script

```
split_date()
{
  ## Assign defaults when no variable names are given on the command line
    sd_1=${2:-SD_YEAR}
    sd_2=${3:-SD_MONTH}
    sd_3=${4:-SD_DAY}


    oldIFS=$IFS        ## save current value of field separator
    IFS='-/. $TAB$NL'  ## new value allows date to be supplied in other formats
    set -- $1          ## place the date into the positional parameters
    IFS=$oldIFS        ## restore IFS
    [ $# -lt 3 ] && return 1  ## The date must have 3 fields

    ## Remove leading zeroes and assign to variables
    eval "$sd_1=\"${1#0}\" $sd_2=\"${2#0}\" $sd_3=\"${3#0}\""
}
```

8.2 is_leap_year—Is There an Extra Day This Year?

Some date calculations need to know whether there are 365 or 366 days in any given year, or whether February has 28 or 29 days.

How It Works

A leap year is a year divisible by 4 but not by 100 unless it is also divisible by 400. This calculation can be done easily in a POSIX shell, but there is a faster method that works even in a Bourne shell.

Usage

```
is_leap_year [YEAR]
```

If no year is given, `is_leap_year` uses `date` to get the current year. The result may be derived from the return code or from the `_IS_LEAP_YEAR` variable:

```
$ is_leap_year && echo yes || echo no ## assuming the current year is 2005
no
$ is_leap_year 2004
$ echo $_IS_LEAP_YEAR
1
$ is_leap_year 2003
$ echo $_IS_LEAP_YEAR
0
```

The Script

Leap years can be determined from the last two, three, or four digits of the number. For example, a year ending in 04 or 08 is always a leap year. Pattern matching reduces the number of possibilities to six, making a look-up table (that works in any Bourne-type shell) the most efficient method.

```
is_leap_year() { ## USAGE: is_leap_year [year]
    ily_year=${1:-$(date +%Y)}
    case $ily_year in
        *0[48] |\
        *[2468][048] |\
        *[13579][26] |\
        *[13579][26]0|\
        *[2468][048]00 |\
        *[13579][26]00 ) _IS_LEAP_YEAR=1
                         return 0 ;;
        *) _IS_LEAP_YEAR=0
           return 1 ;;
    esac
}
```


Notes

Although slower than the preceding script, the canonical method of checking for a leap year would have been a perfectly acceptable script in this book:

```
is_leap_year() {
    ily_year=${1:-`date +%Y`}
    [ $(( $ily_year % 400)) -eq 0 -o \
        \( $(( $ily_year % 4)) -eq 0 -a $(( $ily_year % 100)) -ne 0 \) ] && {
        _IS_LEAP_YEAR=1
        return 0
    } || {
        _IS_LEAP_YEAR=0
        return 1
    }
}
```

8.3 days_in_month—How Many Days Hath September?

One method of determining whether a date is valid uses the number of days in the month. If the day entered is higher than the number of days in the month, the program should reject that date.

How It Works

A simple look-up table will give the number of days in any month except February. For February, the year is also necessary.

Usage

```
_days_in_month [month [year]]  ## result in _DAYS_IN_MONTH
days_in_month [month [year]]   ## result is printed
```

If no month is entered on the command line, `date_vars`, from the `standard-funcs` library in Chapter 1, is used to get the current month and year. If the month is February and no year is given, `date` is used to get the current year.

```
$ days_in_month   ## it is now February 2005
28
$ days_in_month 2 2004
29
```

The Script

```
_days_in_month()
{
    if [ -n "$1" ]  ## If there's a command-line argument...
    then
      dim_m=$1         ## $1 is the month
      dim_y=$2         ## $2 is the year
    else            ## Otherwise use the current date
      date_vars        ## set date variables (from standard-funcs)
      dim_y=$YEAR
      dim_m=$MONTH
    fi
    case ${dim_m#0} in
        ## For all months except February,
        ## a simple look-up table is all that's needed
        9|4|6|11) _DAYS_IN_MONTH=30 ;; ## 30 days hath September...
        1|3|5|7|8|10|12) _DAYS_IN_MONTH=31 ;;

        ## For February, the year is needed in order to check
        ## whether it is a leap year
        2) is_leap_year ${dim_y:-`date +%Y`} &&
             _DAYS_IN_MONTH=29 || _DAYS_IN_MONTH=28 ;;
```

```
        *) return 5 ;;
    esac
}

days_in_month()
{
    _days_in_month $@ && printf %s\n $_DAYS_IN_MONTH
}
```

8.4 date2julian—Calculate the Julian Day Number

8.5 julian2date—Convert Julian Back to Year, Month, and Day

What date will it be 45 days from now? How many days are there until my rent is due? How many days overdue is payment for that invoice I sent? How many days until Christmas? The easy way to calculate dates such as these is to convert them to integers; the dates can then be manipulated by simple addition and subtraction.

How It Works

The Julian Date system used by astronomers was invented by Joseph Scaliger in 1583 and named for his father, Julius Caesar Scaliger, not the Roman emperor (though I expect the connection contributed to the choice of name). The Julian Date is the number of days since noon on January 1, -4712, i.e., January 1, 4713 BC. The time of day is represented by a decimal fraction. Since the Julian day starts at noon, .5 is midnight, and .75 is 6:00 a.m.

For our purposes, we want the day number to refer to the calendar day, from midnight to midnight, so we use the Julian Day (JD) number at noon, which has a time component of 0. There are various formulas for calculating the JD number; this book uses one that was posted to the `comp.unix.shell` newsgroup by Tapani Tarvainen.

Usage

```
date2julian [YEAR-MONTH-DAY]
julian2date JulianDayNumber
```

The conversion functions to and from a Julian Day are mirror images. The first takes an ISO date (YYYY-MM-DD), day, month, and year, and converts them to a single JD integer. The reverse function, `julian2date`, converts the JD number to an ISO date.

```
$ date2julian 1974-10-18
2442339
$ julian2date 2441711
1973-01-28
```

Both these functions are paired with underscore versions (`_date2julian` and `_julian2date`) that just set a variable, but do not print it.

If `date2julian` has no argument, the current day is used; `julian2date` requires an argument.

The Script

```
_date2julian()
{
   ## If there's no date on the command line, use today's date
   case $1 in
        "") date_vars  ## From standard-funcs, Chapter 1
            set -- $TODAY
            ;;
   esac

   ## Break the date into year, month and day
   split_date "$1" d2j_year d2j_month d2j_day || return 2

   ## Since leap years add a day at the end of February,
   ## calculations are done from 1 March 0000 (a fictional year)
   d2j_tmpmonth=$((12 * $d2j_year + $d2j_month - 3))

   ## If it is not yet March, the year is changed to the previous year
   d2j_tmpyear=$(( $d2j_tmpmonth / 12))

   ## The number of days from 1 March 0000 is calculated
   ## and the number of days from 1 Jan. 4713BC is added
   _DATE2JULIAN=$((
        (734 * $d2j_tmpmonth + 15) / 24 -  2 * $d2j_tmpyear + $d2j_tmpyear/4
        - $d2j_tmpyear/100 + $d2j_tmpyear/400 + $d2j_day + 1721119 ))
}

date2julian()
{
    _date2julian "$1" && printf "%s\n" "$_DATE2JULIAN"
}

# ISO date from JD number
_julian2date()
{
    ## Check for numeric argument
    case $1 in
        ""|*[!0-9]*) return 1 ;;
    esac

    ## To avoid using decimal fractions, the script uses multiples.
    ## Rather than use 365.25 days per year, 1461 is the number of days
    ## in 4 years; similarly, 146097 is the number of days in 400 years
    j2d_tmpday=$(( $1 - 1721119 ))
    j2d_centuries=$(( (4 * $j2d_tmpday - 1) / 146097))
```

```
    j2d_tmpday=$(( $j2d_tmpday + $j2d_centuries - $j2d_centuries/4))
    j2d_year=$(( (4 * $j2d_tmpday - 1) / 1461))
    j2d_tmpday=$(( $j2d_tmpday - (1461 * $j2d_year) / 4))
    j2d_month=$(( (10 * $j2d_tmpday - 5) / 306))
    j2d_day=$(( $j2d_tmpday - (306 * $j2d_month + 5) / 10))
    j2d_month=$(( $j2d_month + 2))
    j2d_year=$(( $j2d_year + $j2d_month/12))
    j2d_month=$(( $j2d_month % 12 + 1))

    ## pad day and month with zeros if necessary
    case $j2d_day in ?) j2d_day=0$j2d_day;; esac
    case $j2d_month in ?) j2d_month=0$j2d_month;; esac

    _JULIAN2DATE=$j2d_year-$j2d_month-$j2d_day
}

julian2date()
{
    _julian2date "$1" && printf "%s\n" "$_JULIAN2DATE"
}
```


8.6 dateshift—Add or Subtract a Number of Days

The `date2julian` script lets me convert a date to an integer. Now I need to put that to use and get yesterday's date, or the date one week ago.

How It Works

By encapsulating the conversion to Julian Day, the arithmetic, and the conversion back to ISO date in the `dateshift` function, a single command can return the date at any offset from any given date.

Usage

```
dateshift [YYYY-MM-DD] OFFSET
```

If a date (in ISO format) is not entered on the command line, today's date is used. Therefore, to retrieve yesterday's date:

```
$ dateshift -1  ## at the time of writing, it is October 15, 2004
2004-10-14
```

When is Twelfth Night?

```
$ dateshift 2004-12-25 +12
2005-01-06
```

The Script

```
_dateshift()
{
    case $# in
```

```
        ## If there is only 1 argument, it is the offset
        ## so use today's date
        0|1) ds_offset=${1:-0}
             date_vars
             ds_date=$TODAY
             ;;
        ## ...otherwise the first argument is the date
        ) ds_date=$1
           ds_offset=$2
           ;;
    esac
    while :
    do
       case $ds_offset in
           0*|+*) ds_offset=${ds_offset#?} ;; ## Remove leading zeros or plus signs
           -*) break ;; ## Negative value is OK; exit the loop
           "") ds_offset=0; break ;;         ## Empty offset equals 0; exit loop
           *[!0-9]*) return 1 ;;             ## Contains non-digit; return with error
           *) break ;;                       ## Let's assume it's OK and continue
       esac
    done
    ## Convert to Julian Day
    _date2julian "$ds_date"
    ## Add offset and convert back to ISO date
    _julian2date $(( $_DATE2JULIAN + $ds_offset ))
    ## Store result
    _DATESHIFT=$_JULIAN2DATE
}

dateshift()
{
    _dateshift "$@" && printf "%s\n" "$_DATESHIFT"
}
```

Notes

I find it convenient to have separate commands for the commonly used calculations:

```
_yesterday()
{
    _date2julian "$1"
    _julian2date $(( $_DATE2JULIAN - 1 ))
    _YESTERDAY=$_JULIAN2DATE
}

_tomorrow()
{
    _date2julian "$1"
```

```
    _julian2date $(( $_DATE2JULIAN + 1 ))
    _TOMORROW=$_JULIAN2DATE
}
```

8.7 diffdate—Find the Number of Days Between Two Dates

How many days do I have until the next deadline? How many days are between my birthday and Christmas?

How It Works

This is another encapsulation of conversion and arithmetic, using two dates.

Usage

```
diffdate YYYY-MM-DD [YYYY-MM-DD]
```

If only one date is given, the current date is used as the first date:

```
$ diffdate 2004-12-25 ## today is October 15, 2004
71
```

If the second date is earlier than the first, the result will be negative:

```
$ diffdate 2005-03-22 1998-10-18
-2347
```

The Script

This script simply converts the two dates (or one date and today's date) and subtracts one from the other.

```
_diffdate()
{
    case $# in
        ## If there's only one argument, use today's date
        1) _date2julian $1
           dd2=$_DATE2JULIAN
           _date2julian
           dd1=$_DATE2JULIAN
           ;;
        2) _date2julian "$1"
           dd1=$_DATE2JULIAN
           _date2julian "$2"
           dd2=$_DATE2JULIAN
           ;;
    esac
    _DIFFDATE=$(( $dd2 - $dd1 ))
}
```

```
diffdate()
{
    _diffdate "$@" && printf "%s\n" "$_DIFFDATE"
}
```

8.8 day_of_week—Find the Day of the Week for Any Date

I need to know whether a certain date is a business day, or whether it is on the weekend.

How It Works

This is another use for the Julian Day. If you add 1 to the JD and divide by 7, the remainder is the day of the week, counting from Sunday as 0. A look-up table converts the number to a name.

Usage

```
day_of_week [YYYY-MM-DD]
dayname N
```

To find which day of the week Christmas is on, use this:

```
$ day_of_week 2005-12-25
0
```

Christmas Day, 2005 falls on a Sunday. To convert the number to the name of the day, `dayname` uses a simple look-up table:

```
$ dayname 0
Sunday
```

The `dayname` function will also complete the name of the day if you give it an abbreviation:

```
$ dayname wed
Wednesday
```

If `day_of_week` is called without an argument, it uses the current day's date.

The Script

```
_day_of_week()
{
    _date2julian "$1"
    _DAY_OF_WEEK=$(( ($_DATE2JULIAN + 1) % 7 ))
}

day_of_week()
{
    _day_of_week "$1" && printf "%s\n" "$_DAY_OF_WEEK"
}

## Dayname accepts either 0 or 7 for Sunday, 2-6 for the other days
## or checks against the first three letters, in upper or lower case
```

```
_dayname()
{
    case ${1} in
        0|7|[Ss][Uu][Nn]*) _DAYNAME=Sunday ;;
        1|[Mm][Oo][nN]*) _DAYNAME=Monday ;;
        2|[Tt][Uu][Ee]*) _DAYNAME=Tuesday ;;
        3|[Ww][Ee][Dd]*) _DAYNAME=Wednesday ;;
        4|[Tt][Hh][Uu]*) _DAYNAME=Thursday ;;
        5|[Ff][Rr][Ii]*) _DAYNAME=Friday ;;
        6|[Ss][Aa][Tt]*) _DAYNAME=Saturday ;;
        *) return 5 ;; ## No match; return an error
    esac
}

dayname()
{
    _dayname "$@" && printf "%s\n" "$_DAYNAME"
}
```


8.9 display_date—Show a Date in Text Format

The ISO date format is great for computers, but it's not as easy for people to read. We need a way to display dates in a more user-friendly way.

How It Works

Look-up tables convert the day of the week and the month to their respective names. The user can supply different strings to vary the format used.

Usage

```
display_date [-f FMT] [YYYY-MM-DD]
```

If no format is supplied, the default is used:

```
$ display_date 2005-02-14
Wednesday, 14 February 2005
```

There are only four format strings defined at the moment (WMdy, dMy, Mdy, and WdMy), but you can easily add more. Here are examples of all four formats:

```
$ for fmt in  WMdy dMy Mdy WdMy
> do
>   date=$(( $RANDOM % 100 + 1950 ))-$(( $RANDOM % 12 ))-$(( $RANDOM % 28 ))
>   display_date -f "$fmt" "$date"
> done
Thursday, July 12, 1998
14 November 1964
January 21, 2009
Monday, 18 January 2018
```

The Script

```
display_date()
{
    dd_fmt=WdMy  ## Default format

    ## Parse command-line options for format string
    OPTIND=1
    while getopts f: var
    do
      case $var in
          f) dd_fmt=$OPTARG ;;
      esac
    done
    shift $(( $OPTIND - 1 ))

    ## If there is no date supplied, use today's date
    case $1 in
        "") date_vars ## Function from standard-funcs in Chapter 1
            set -- $TODAY
            ;;
    esac

    split_date "$1" dd_year dd_month dd_day || return 2

    ## Look up long names for day and month
    _day_of_week "$1"
    _dayname $_DAY_OF_WEEK
    _monthname $dd_month

    ## Print date according to format supplied
    case $dd_fmt in
        WMdy) printf "%s, %s %d, %d\n" "$_DAYNAME" "$_MONTHNAME" \
                    "$dd_day" "$dd_year" ;;
        dMy)  printf "%d %s %d\n" "$dd_day" "$_MONTHNAME" "$dd_year" ;;
        Mdy)  printf "%s %d, %d\n" "$_MONTHNAME" "$dd_day" "$dd_year" ;;
        WdMy|*) printf "%s, %d %s %d\n" "$_DAYNAME" "$dd_day" \
                      "$_MONTHNAME" "$dd_year" ;;
    esac
}

## Set the month number from 1- or 2-digit number, or the name
_monthnum()
{
    case ${1#0} in
         1|[Jj][aA][nN]*) _MONTHNUM=1 ;;
         2|[Ff][Ee][Bb]*) _MONTHNUM=2 ;;
         3|[Mm][Aa][Rr]*) _MONTHNUM=3 ;;
```

```
         4|[Aa][Pp][Rr]*) _MONTHNUM=4 ;;
         5|[Mm][Aa][Yy]*) _MONTHNUM=5 ;;
         6|[Jj][Uu][Nn]*) _MONTHNUM=6 ;;
         7|[Jj][Uu][Ll]*) _MONTHNUM=7 ;;
         8|[Aa][Uu][Gg]*) _MONTHNUM=8 ;;
         9|[Ss][Ee][Pp]*) _MONTHNUM=9 ;;
        10|[Oo][Cc][Tt]*) _MONTHNUM=10 ;;
        11|[Nn][Oo][Vv]*) _MONTHNUM=11 ;;
        12|[Dd][Ee][Cc]*) _MONTHNUM=12 ;;
        *) return 5 ;;
    esac
}

monthnum()
{
   _monthnum "$@" && printf "%s\n" "$_MONTHNUM"
}

## Set the month name from 1- or 2-digit number, or the name
_monthname()
{
    case ${1#0} in
         1|[Jj][aA][nN]) _MONTHNAME=January ;;
         2|[Ff][Ee][Bb]) _MONTHNAME=February ;;
         3|[Mm][Aa][Rr]) _MONTHNAME=March ;;
         4|[Aa][Pp][Rr]) _MONTHNAME=April ;;
         5|[Mm][Aa][Yy]) _MONTHNAME=May ;;
         6|[Jj][Uu][Nn]) _MONTHNAME=June ;;
         7|[Jj][Uu][Ll]) _MONTHNAME=July ;;
         8|[Aa][Uu][Gg]) _MONTHNAME=August ;;
         9|[Ss][Ee][Pp]) _MONTHNAME=September ;;
        10|[Oo][Cc][Tt]) _MONTHNAME=October ;;
        11|[Nn][Oo][Vv]) _MONTHNAME=November ;;
        12|[Dd][Ee][Cc]) _MONTHNAME=December ;;
        *) return 5 ;;
    esac
}

monthname()
{
    _monthname "$@" && printf "%s\n" "${_MONTHNAME}"
}
```

8.10 parse_date—Decipher Various Forms of Date String

Dates come in many different formats, some straightforward, some ambiguous. Some are easy to parse, others are not. Some put the month before the date, some put it after. Some use numbers for the month, some use the name of the month. Some separate the components with spaces, some with slashes. Out of these many formats, is there a way to extract a coherent date?

How It Works

Designed to give the widest possible latitude in date entry, `parse_date` can be told to expect a specific format, or it can chew on the input and, with a bit of luck, spit out a valid ISO date. The three options tell `parse_date` to interpret the fields as YMD, DMY, or MDY. If no option is given, the script will attempt to figure out what is meant.

Usage

```
parse_date [-eiu] DATE
```

The options set the order that `parse_date` assigns the fields: `-e` uses day-month-year, `-u` uses month-day-year, and `-i` uses year-month-day (think **E**nglish, **U**S, and **I**nternational).

The month may be entered as a number from 1 to 12, or as the name of the month; the fields may be separated by whitespace, periods, hyphens, or slashes. The result is printed as an ISO date, year-month-day:

```
$ parse_date -e 12.4.2001
2001-04-12
$ parse_date -u 12.4.2001
2001-12-04
$ parse_date 12.apr.2001
2001-04-12
```

Invalid dates are caught, and an error is returned:

```
$ parse_date -u 2004-12-10; echo $?
2
$ parse_date -i 12.4.2001; echo $?
2
```

There are a few shortcuts. Today, yesterday, and tomorrow can be used instead of the actual dates; these can be represented by a period, hyphen, and plus sign, respectively:

```
$ parse_date .
2004-10-15
$ parse_date +
2004-10-16
$ parse_date -
2004-10-14
```

A date 30 days from now may be entered as +30; one week ago can be entered as -7:

```
$ parse_date +30
2004-11-14
$ parse_date -7
2004-10-08
```

Ambiguous dates return an error (but an option can remove the ambiguity):

```
$ parse_date 2.3.2001 || echo ambigous >&2
ambigous
$ parse_date -e 2.3.2001 || echo ambigous
2001-03-02
$ parse_date -u 2.3.2001 || echo ambigous
2001-02-03
```

The Script

```
_parse_date()
{
    ## Clear variables
    _PARSE_DATE=
    pd_DMY=
    pd_day=
    pd_month=
    pd_year=

    ## If no date is supplied, read one from the standard input
    case $1 in
        "") [ -t 0 ] && printf "Date: " >&2 ## Prompt only if connected to a terminal
            read pd_date
            set -- $pd_date
            ;;
    esac

    ## Accept yesterday, today and tomorrow as valid dates
    case $1 in
        yes*|-)
            _yesterday && _PARSE_DATE=$_YESTERDAY
            return
            ;;
        tom*|+)
            _tomorrow  && _PARSE_DATE=$_TOMORROW
            return
            ;;
        today|.)
            date_vars && _PARSE_DATE=$TODAY
            return
            ;;
        today*|\
```

```
    .[-+1-9]* |\
    [-+][1-9]* )
        pd_=${1#today}
        pd_=${pd_#[-+]}
        _dateshift $pd_ && _PARSE_DATE=$_DATESHIFT
        return
        ;;
esac

## Parse command-line options for date format
OPTIND=1
while getopts eiu var
do
  case $var in
      e) pd_DMY=dmy ;;
      i) pd_DMY=ymd ;;
      u) pd_DMY=mdy ;;
  esac
done
shift $(( $OPTIND - 1 ))

## Split date into the positional parameters
oldIFS=$IFS
IFS='/-. $TAB$NL'
set -- $*
IFS=$oldIFS

## If date is incomplete, use today's information to complete it
if [ $# -lt 3 ]
then
  date_vars
  case $# in
      1) set -- $1 $MONTH $YEAR ;;
      2) set -- $1 $2 $YEAR ;;
  esac
fi

case $pd_DMY in
   ## Interpret date according to format if one has been defined
    dmy) pd_day=${1#0}; pd_month=${2#0}; pd_year=$3 ;;
    mdy) pd_day=${2#0}; pd_month=${1#0}; pd_year=$3 ;;
    ymd) pd_day=${3#0}; pd_month=${2#0}; pd_year=$1 ;;

    ## Otherwise make an educated guess
    *) case $1--$2-$3 in
           [0-9][0-9][0-9][0-9]*-*)
                     pd_year=$1
                     pd_month=$2
```

```
                        pd_day=$3
                        ;;
                *--[0-9][0-9][0-9][0-9]*-*) ## strange place
                        pd_year=$2
                        _parse_dm $1 $3
                        ;;
                *-[0-9][0-9][0-9][0-9]*)
                        pd_year=$3
                        _parse_dm $1 $2
                        ;;
                *) return 5 ;;
            esac

            ;;
    esac

    ## If necessary, convert month name to number
    case $pd_month in
        [JjFfMmAaSsOoNnDd]*) _monthnum "$pd_month" || return 4
                             pd_month=$_MONTHNUM
                             ;;
        *[!0-9]*) return 3 ;;
    esac

    ## Quick check to eliminate invalid day or month
    [ "${pd_month:-99}" -gt 12 -o "${pd_day:-99}" -gt 31 ] && return 2

    ## Check for valid date, and pad day and month if necessary
    _days_in_month $pd_month $pd_year
    case $pd_day in ?) pd_day=0$pd_day;; esac
    case $pd_month in ?) pd_month=0$pd_month;; esac
    [ ${pd_day#0} -le $_DAYS_IN_MONTH ] &&
                _PARSE_DATE="$pd_year-$pd_month-$pd_day"
}

parse_date()
{
    _parse_date "$@" && printf "%s\n" "$_PARSE_DATE"
}

## Called by _parse_date to determine which argument is the month
## and which is the day
_parse_dm()
{
    ## function requires 2 arguments; more will be ignored
    [ $# -lt 2 ] && return 1
```

```
    ## if either argument begins with the first letter of a month
    ## it's a month; the other argument is the day
    case $1 in
        [JjFfMmAaSsOoNnDd]*)
             pd_month=$1
             pd_day=$2
             return
             ;;
    esac
    case $2 in
        [JjFfMmAaSsOoNnDd]*)
             pd_month=$2
             pd_day=$1
             return
             ;;
    esac

    ## return error if either arg contains non-numbers
    case $1$2 in *[!0-9]*) return 2;; esac

    ## if either argument is greater than 12, it is the day
    if [ $1 -gt 12 ]
    then
      pd_day=$1
      pd_month=$2
    elif [ ${2:-0} -gt 12 ]
    then
      pd_day=$2
      pd_month=$1
    else
      pd_day=$1
      pd_month=$2
      return 1 ## ambiguous
    fi
}
```

8.11 valid_date—Where Was I on November 31st?

When you already have a well-formed date, `parse_date` is overkill for verification. A simpler script will suffice.

How It Works

If you convert a date to a Julian Day, then convert it back, the values will not be the same if the date is not valid:

```
$ julian2date $(date2julian 2004-12-32)
2005-01-01
```

This function does just that, and fails if the dates do not match.

Usage

```
valid_date YEAR-MONTH-DAY
```

There are only 30 days in April, so April 31 will not be valid:

```
$ valid_date 2005-04-31 && echo OK || echo Invalid date
Invalid date
$ valid_date 2005-04-30 && echo OK || echo Invalid date
OK
```

The Script

```
valid_date()
{
    _date2julian "$1" || return 8
    _julian2date $_DATE2JULIAN || return 7
    [ $_JULIAN2DATE = $1 ]
}
```


Summary

Many of the functions presented here can be achieved with the GNU version of the `date` command. Others, such as `parse_date`, have more flexibility. These functions will often be faster than `date`, but not always by much. They do speed up script writing by tucking many details away behind the scenes.

I could include many more date functions, some that I haven't yet separated into their own functions because they are small additions to scripts already here, and they can easily be entered in-line with the rest of a script. Some are here because I use them a lot, or because they answer newsgroup FAQs.

An example that fits into both categories is `yesterday`. I have a cron job that runs every night at midnight and archives certain files. I want to date them with the preceding day's date. Having a one-word command made the script easier to write, and the result easier to read.

I have thought about writing a script that would accept a formatting string for printing the date, a `display_date` on steroids, but it would be far more work to write than it warrants. I can use GNU `date` when I need that facility.