UPA 4 - Calendar Year Program

In this program, you will extend the calendar month program (from Unit 3 UPA) to display a complete calendar year.

The Problem

The problem is to display a calendar year for any year between 1800 and 2099, inclusive. The format of the displayed year should be as seen below:

```
2015

January                  February                 March
                1  2  3   1  2  3  4  5  6  7      1  2  3  4  5  6  7
 4  5  6  7  8  9 10      8  9 10 11 12 13 14      8  9 10 11 12 13 14
11 12 13 14 15 16 17     15 16 17 18 19 20 21     15 16 17 18 19 20 21
18 19 20 21 22 23 24     22 23 24 25 26 27 28     22 23 24 25 26 27 28
25 26 27 28 29 30 31                              29 30 31


April                    May                      June
          1  2  3  4               1  2             1  2  3  4  5  6
 5  6  7  8  9 10 11      3  4  5  6  7  8  9      7  8  9 10 11 12 13
12 13 14 15 16 17 18     10 11 12 13 14 15 16     14 15 16 17 18 19 20
19 20 21 22 23 24 25     17 18 19 20 21 22 23     21 22 23 24 25 26 27
26 27 28 29 30           24 25 26 27 28 29 30     28 29 30
                         31


July                     August                   September
          1  2  3  4                  1              1  2  3  4  5
 5  6  7  8  9 10 11      2  3  4  5  6  7  8      6  7  8  9 10 11 12
12 13 14 15 16 17 18      9 10 11 12 13 14 15     13 14 15 16 17 18 19
19 20 21 22 23 24 25     16 17 18 19 20 21 22     20 21 22 23 24 25 26
26 27 28 29 30 31        23 24 25 26 27 28 29     27 28 29 30
                         30 31


October                  November                 December
                1  2  3   1  2  3  4  5  6  7      1  2  3  4  5
 4  5  6  7  8  9 10      8  9 10 11 12 13 14      6  7  8  9 10 11 12
11 12 13 14 15 16 17     15 16 17 18 19 20 21     13 14 15 16 17 18 19
18 19 20 21 22 23 24     22 23 24 25 26 27 28     20 21 22 23 24 25 26
25 26 27 28 29 30 31     29 30                    27 28 29 30 31
```

Problem Analysis

The computational issues for this problem are similar to the calendar month program of Chapter 3. We need an algorithm for computing the first day of a given month for years 1800–2099. However, since the complete year is being displayed, only the day of the week for January 1st of the given year needs be computed—the rest of the days follow from knowing the number of days in each month (including February for leap years). The algorithm previously developed to

display a calendar month, however, is not relevant for this program. Instead, the information will first be stored in a data structure allowing for the months to be displayed three across.

Program Design

Meeting the Program Requirements

You will develop and implement an algorithm that displays the calendar year as shown on the previous page. You shall request the user to enter the four-digit year to display, with appropriate input error checking.

Data Description

The program needs to represent the year entered, whether it is a leap year, the day of the week for January 1st of the year, and the number of days in each month (accounting for leap years). The names of each of the twelve months will also be stored for display in the calendar year. Given this information, the calendar year can be appropriately constructed and displayed.

You will make use of nested lists for representing the calendar year. The data structure will start out as an empty list and will be built incrementally as each new calendar month is computed. The list structures for the calendar year and calendar month are given below,

```
calendar_year = [ [calendar_month], [calendar_month], etc.] ]

calendar_month = [ week_1, week_2, . . ., week_k ]
```

Each italicized month is represented as a list of four to six strings, with each string storing a week of the month to be displayed (or a blank line for alignment purposes).

FEBRUARY 2015

Sun	Mon	Tues	Wed	Thur	Fri	Sat	
1	2	3	4	5	6	7	*4 lines*
8	9	10	11	12	13	14	
15	16	17	18	19	20	21	
22	23	24	25	26	27	28	

MAY 2015

Sun	Mon	Tues	Wed	Thur	Fri	Sat	
					1	2	*6 lines*
3	4	5	6	7	8	9	
10	11	12	13	14	15	16	
17	18	19	20	21	22	23	
24	25	26	27	28	29	30	
31							

The strings are formatted to contain all the spaces needed for proper alignment when displayed. For example, since the first week of May 2015 begins on a Friday, the string value for this week would be,

The complete representation for the calendar year 2015 follows, with the details shown for the months of February and May.

```
[[ January ],
 [ ' 1 2 3 4 5 6 7', ' 8 9 10 11 12 13 14',                    February
   ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28'],
 [ March ],
 [ April ],
 [ '                1 2', ' 3 4 5 6 7 8 9',                    May
   ' 10 11 12 13 14 15 16', ' 17 18 19 20 21 22 23',
   ' 24 25 26 27 28 29 30', ' 31                  '],
 [ June ],
 [ July ],
 [ August ],
 [ September ],
 [ October ],
 [ November ],
 [ December ]]
```

(Typically, yearly calendars combine the one or two remaining days of the month on the sixth line of a calendar month onto the previous week. We shall not do that in this program, however.)

Algorithmic Approach

You will make use of the “day of the week” algorithm that you previously used. For this program, however, the only date for which the day of the week needs to be determined is January 1 of a given year. Thus, the original day of the week algorithm can be simplified by removing variable `day` and replacing its occurrence on line 6 with 1, as seen below.

To determine the day of the week for January 1 of a given **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
   if **century_digits** equals 20, then add 6 to **value**.
5. If **year** is not a leap year then add 1 to **value**.
6. Set **value** equal to (**value** + 1) mod 7.
7. If **value** is equal to 1 (Sunday), 2 (Monday), … 0 (Saturday).

Overall Program Steps

The overall steps in this program design are illustrated here:

Program Implementation and Testing

Stage 1—Determining the Day of the Week (for January 1st)

You will first write and test the code for determining the day of the week for January 1st of a given year.

Enter the Stage 1 Code from the sample on the next page.

```
1  # Calendar Year Program (Stage 1)
2
3  # initialization
4  terminate = False
5
6  # prompt for years until quit
7  while not terminate:
8
9      # get year
10     year = int(input('Enter year (yyyy) (-1 to quit): '))
11
12     while (year < 1800 or year > 2099) and year != -1:
13         year = int(input('INVALID - Enter year(1800-2099): '))
14
15     if year == -1:
16         terminate = True
17     else:
18         # determine if leap year
19         if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
20             leap_year = True
21         else:
22             leap_year = False
23
24         # determine day of the week
25         century_digits = year // 100
26         year_digits = year % 100
27
28         value = year_digits + (year_digits // 4)
29
30         if century_digits == 18:
31             value = value + 2
32         elif century_digits == 20:
33             value = value + 6
34
35         # leap year check
36         if not leap_year:
37             value = value + 1
38
39         # determine first day of month for Jan 1
40         first_day_of_month = (value + 1) % 7
41
42         print('Day of week is:', first_day_of_month)
```

Notes:

Line 4 initializes Boolean flag `terminate` to `False`. If the user enters `-1` for the year (in lines 10-13), `terminate` is set to `True` and the while loop at line 7 terminates, thus terminating the program. If a valid year is entered, lines 19-42 are executed.

Lines 19-22 determine if the year is a leap year using the same code as in the calendar month program, assigning Boolean variable `leap_year` accordingly. Lines 25-40 implement the simplified day of the week algorithm for determining the day of the week for January 1 of a given year in Figure 4.17, with the result displayed on line 42.

Stage 1—Testing

Test the Stage 1 code before continuing. A sample test run of this stage of the program is shown here:

```
Enter year (yyyy) (-1 to quit): 1800
Day of week is: 4
Enter year (yyyy) (-1 to quit): 1900
Day of week is: 2
Enter year (yyyy) (-1 to quit): 1984
Day of week is: 1
Enter year (yyyy) (-1 to quit): 1985
Day of week is: 3
Enter year (yyyy) (-1 to quit): 3000
INVALID - Enter year(1800-2099): 2000
Day of week is: 0
Enter year (yyyy) (-1 to quit): 2009
Day of week is: 5
Enter year (yyyy) (-1 to quit): -1
>>>
```

The following table displays possible test cases used for the program.

Calendar Month	Expected Results first day of month	Actual Results first day of month	Evaluation
January 1800	4 (Wednesday)	4	Passed
January 1900	2 (Monday)	2	Passed
January 1984	1 (Sunday)	1	Passed
January 1985	3 (Tuesday)	3	Passed
January 2000	0 (Saturday)	0	Passed
January 2009	5 (Thursday)	5	Passed

If all test cases pass, you can move on to the next stage of program development.

Stage 2—Constructing the Calendar Year Data Structure

Next you will develop the part of the program that constructs the data structure holding all of the calendar year information to be displayed. The data structure begins empty and is incrementally built, consisting of nested lists.

(Code follows, on next two pages)

```
# Calendar Year Program (Final Version)

# initialization
terminate = False
days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_names = ('January', 'February', 'March', 'April', 'May', 'June'
               'July', 'August', 'September', 'October', 'November',
               'December')

calendar_year = []
month_separator = format(' ', '8')
blank_week = format(' ', '21')
blank_col = format(' ', '3')

# prompt for years until quit
while not terminate:

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or
            (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        century_digits = year // 100
        year_digits = year % 100
        value = year_digits + (year_digits // 4)

        if century_digits == 18:
            value = value + 2
        elif century_digits == 20:
            value = value + 6

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_current_month = (value + 1) % 7

        # construct calendar for all 12 months
        for month_num in range(12):
            month_name = month_names[month_num]

            # init for new month
            current_day = 1
            if first_day_of_current_month == 0:
                starting_col = 7
            else:
                starting_col = first_day_of_current_month

            current_col = 1
            calendar_week = ''
            calendar_month = []

```

```
66              # add any needed leading space for first week of month
67              while current_col < starting_col:
68                  calendar_week = calendar_week + blank_col
69                  current_col = current_col + 1
70
71              # store month as separate weeks
72              if (month_name == 'February') and leap_year:
73                  num_days_this_month = 29
74              else:
75                  num_days_this_month = days_in_month[month_num]
76
77              while current_day <= num_days_this_month:
78
79                  # store day of month in field of length 3
80                  calendar_week = calendar_week + \
81                     format(str(current_day),'>3')
82
83                  # check if at last column of displayed week
84                  if current_col == 7:
85                      calendar_month = calendar_month + [calendar_week]
86                      calendar_week = ''
87                      current_col = 1
88                  else:
89                      current_col = current_col + 1
90
91                  # increment current day
92                  current_day = current_day + 1
93
94              # fill out final row of month with needed blanks
95              calendar_week = calendar_week + \
96               blank_week[0:(7-current_col+1) * 3]
97              calendar_month = calendar_month + [calendar_week]
98
99              # reset values for next month
100             first_day_of_current_month = current_col
101             calendar_year = calendar_year + [calendar_month]
102             calendar_month = []
103
104     print(calendar_year)
105
106     #reset for another year
107     calendar_year = []
```

Notes:

Lines 4–14 perform the required initialization. Tuples `days_in_month` and `month_ names` have been added to the program to store the number of days for each month (with February handled as an exception) and the month names. On line 11, `calendar_year` is initialized to the empty list. It will be constructed month-by-month for the twelve months of the year. There is the need for strings of blanks of various lengths in the program, initialized as `month_separator`, `blank_week`, and `blank_col` (lines 12–14). The `calendar_ year` data structure will contain all the space characters needed for the calendar months to be

properly displayed. Therefore, there will be no need to develop code that determines how each month should be displayed as in the calendar month program. The complete structure will simply be displayed row by row.

Lines 17–49 are the same as the first stage of the program for determining the day of the week of a given date. Once the day of the week for January 1st of the given year is known, the days of the week for all remaining dates simply follow. Thus, there is no need to calculate the day of the week for any other date.

Line 52 begins the for loop for constructing each of the twelve months. On line 53, the month name is retrieved from tuple `month_names` and assigned to `month_name`. Variable `current_day`, holding the current day of the month, is initialized to `1` for the new month (line 56). In lines 57–60, `first_day_of_current_month`, determined by the day of the week algorithm, is converted to the appropriate column number. Thus, since `0` denotes Saturday, if `first_day_of_current_month` equals `0`, `starting_col` is set to 7. Otherwise, `starting_col` is set to `first_day_of_current_month` (e.g., if `first_day_of_current_month` is `1`, then `starting_col` is set to 1).

In lines 62–64, the initialization for a new month finishes with the reassignment of `current_col`, `calendar_week`, and `calendar_month`. Each calendar week of a given month is initially assigned to the empty string, with each date appended one-by-one. Variable `current_col` is used to keep track of the current column (day) of the week, incremented from `0` to 6. Since the first day of the month can fall on any day of the week, the first week of any month may contain blank (“skipped”) columns. This includes the columns from `current_col` up to but not including `starting_col`. The while loop in lines 67–69 appends any of these skipped columns to empty string `calendar_week`.

Lines 72–75 assign `num_days_this_month` to the number of days stored in tuple `days_in_month`. The exception for February, based on whether the year is a leap year or not, is handled as a special case. The while loop at line 77 increments variable `current_day` from `1` to the number of days in the month. In lines 80–81 each date is appended to `calendar_week` right-justified as a string of length three by use of the format function. Thus, a single-digit date will be appended with two leading blanks, and a double-digit date with one leading blank so that the columns of dates align.

For each new date appended to `calendar_week`, a check is made on line 84 as to whether the end of the week has been reached. If the last column of the calendar week has been reached (when `column_col` equals 7) then the constructed `calendar_week` string is appended to the `calendar_month` (line 85). In addition, `calendar_week` is re-initialized to the empty string, and `current_col` is reset to 1 (lines 86–87). If the last column of the calendar week has not yet been reached, then `current_col` is simply incremented by 1 (line

89). Then, on line 92, variable `current_day` is incremented by 1, whether or not a new week is started.

When the while loop (at line 77) eventually terminates, variable `current_week` holds the last week of the constructed month. Therefore, as with the first week of the month, the last week may contain empty columns. This is handled by lines 95–97. Before appending `calendar_week` to `calendar_month`, any remaining unfilled columns are appended to it (the reason that these final columns must be blank-filled is because months are displayed side-by-side, and therefore are needed to keep the whole calendar properly aligned),

```
calendar_week = calendar_week + blank_week[0:(7-current_col+1) * 3]
```

Thus, the substring of `blank_week` produced will end up as an empty string if the value of `current_col` is `6` (for Saturday, the last column) as it should. Line 100 sets variable `first_day_ of_current_month` to `current_col` since `current_col` holds the column value of the next column that *would have been* used for the current month, and thus is the first day of the following month. On line 101, the completed current month is appended to list `calendar_year`. And on line 102, `calendar_month` is reset to an empty list in anticipation of the next month to be constructed. Finally, on line 104, the complete `calendar_year` list is displayed. Because the program prompts the user for other years to be constructed and displayed, the `calendar_year` list is reset to the empty list (line 107).

Stage 2 - Testing

Run the program, and you should find that the program terminates with an error on line 53,

```
Enter year (yyyy) (-1 to quit): 2015
Traceback (most recent call last):
  File "C:\My Python Programs\CalendarYearStage2.py", line 54, in <module>
    month_name = month_names[month_num]
IndexError: tuple index out of range
```

This line is within the for loop at line 52,

```
for month_num in range(12):
    month_name = month_names[month_num]
```

For some reason, index variable `month_num` is out of range for tuple `month_names`. Look at the final value of `month_num` by typing the variable name into the Python shell,

```
>>> month_num
11
```

Since `month_names` has index values 0–11 (since of length 12), an index value of 11 should not be out of range. How, then, can this index out of range error happen? Just to make sure that `month_ names` has the right values, we display its length,

```
>>>len(month_names)
11
```

This is not right! The tuple `month_names` should contain all twelve months of the year. That is the way it was initialized on line 7, and tuples, unlike lists, cannot be altered, they are immutable. This does not seem to make sense. To continue our investigation, we display the value of the tuple,

```
>>> month_names
('January', 'February', 'March', 'April', 'May', 'JuneJuly',
'August', 'September', 'October', 'November', 'December')
>>>
```

Now we see something that doesn't look right. Months `June` and `July` are concatenated into one string value `'JuneJuly'` making the length of the tuple 11, and not 12 (as we discovered). *That* would explain why the index out of range error occurred.

What, then, is the problem. Why were the strings `'June'` and `'July'` concatenated? We need to look at the line of code that creates this tuple,

```
month_names 5('January', 'February', 'March', 'April', 'May', 'June'
              'July', 'August', 'September', 'October', 'November', 'December')
```

It looks OK. Strings `'June'` and `'July'` were written as separate strings. We then decide to count the number of items in the tuple. Since items in tuples and lists are separated by commas, we count the number of items between the commas. We count the items up to `'May'`, which is five items as it should be, then `'June'`, which is six items . . . ah, there is no comma after the string `'June'`! *That* must be why strings `'June'` and `'July'` were concatenated, and thus the source of the index out of range error. We try to reproduce this in the shell,

```
>>> 'June' 'July'
'JuneJuly'
```

That's it! We have found the problem and should feel good about it.

So… make this correction in your code.

Then re-execute the program.

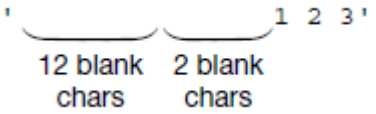
After making the correction and re-executing the program, you should get the following results:

```
Enter year (yyyy) (-1 to quit): 2015
[['              1  2  3', '  4  5  6  7  8  9 10', ' 11 12 13 14 15 16 17', ' 1
8 19 20,21 22 23 24', ' 25 26 27 28 29 30 31','                       '], ['  1
2  3  4  5  6  7', '  8  9 10 11 12 13 14', ' 15 16 17 18 19 20 21', ' 22 23 24
25 26 27 28', '                     '], ['  1  2  3  4  5  6  7', '  8  9 10 11
12 13 14', ' 15 16 17 18 19 20 21', ' 22 23 24 25 26 27 28', ' 29 30 31
   '], ['           1  2  3  4', '  5  6  7  8  9 10 11', ' 12 13 14 15 16 17 18
', ' 19 20 21 22 23 24 25', ' 26 27 28 29 30     '], ['                  1  2',
'   3  4  5  6  7  8  9', ' 10 11 12 13 14 15 16', ' 17 18 19 20 21 22 23', ' 24
25 26 27 28 29 30', ' 31                  '], ['    1  2  3  4  5  6', '  7  8
 9 10 11 12 13', ' 14 15 16 17 18 19 20', ' 21 22 23 24 25 26 27', ' 28 29 30
         '], ['           1  2  3  4', '  5  6  7  8  9 10 11', ' 12 13 14 15 16
 17 18', ' 19 20 21 22 23 24 25', ' 26 27 28 29 30 31    '], ['
  1', '  2  3  4  5  6  7  8', '  9 10 11 12 13 14 15', ' 16 17 18 19 20 21 22',
 ' 23 24 25 26 27 28 29', ' 30 31               '], ['        1  2  3  4  5', '
 6  7  8  9 10 11 12', ' 13 14 15 16 17 18 19', ' 20 21 22 23 24 25 26', ' 27 28
 29 30         '], ['              1  2  3', '  4  5  6  7  8  9 10', ' 11 12 13
 14 15 16 17', ' 18 19 20 21 22 23 24', ' 25 26 27 28 29 30 31', '
       '], ['  1  2  3  4  5  6  7', '  8  9 10 11 12 13 14', ' 15 16 17 18 19 2
0 21', ' 22 23 24 25 26 27 28', ' 29 30              '], ['        1  2  3  4
5', '  6  7  8  9 10 11 12', ' 13 14 15 16 17 18 19', ' 20 21 22 23 24 25 26', '
 27 28 29 30 31      ']]
Enter year (yyyy) (-1 to quit):
```

We can see if the output looks like the structure that we expect. The first item in the list, the structure for the month of January, is as follows,

```
[['              1 2 3', ' 4 5 6 7 8 9 10', ' 11 12 13 14 15 16 17',
' 18 19 20 21 22 23 24', ' 25 26 27 28 29 30 31', '                      ']
```

In checking against available calendar month calculators, we see that the first day of the month for January 2015 is a Thursday. Thus, the first week of the month should have four skipped days, followed by 1, 2, and 3 each in a column width of 3. We find that there are fourteen blank characters in the first line. The first twelve are for the four skipped columns, and the last two are for the right-justified string ‘1’ in the column of the first day of the month,



Since there are five weeks in the month, there should be one extra “blank week” at the end of the list to match the vertical spacing of all other months. We see, in fact, that the last (sixth) string is a string of blanks.

Since the `calendar_year` structure looks correct, we now develop the final stage of the program that displays the complete calendar year.

Stage 3—Displaying the Calendar Year Data Structure

Update the code based on the following. See notes for details.

```
# Calendar Year Program (Final Version)

# initialization
terminate = False
days_in_month = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

month_names = ('January', 'February', 'March', 'April', 'May', 'June',
               'July', 'August', 'September', 'October', 'November',
               'December')

calendar_year = []
month_separator = format(' ', '8')
blank_week = format(' ', '21')
blank_col = format(' ', '3')

# prompt for years until quit
while not terminate:

    # program greeting
    print ('This program will display a calendar year for a given year')

    # get year
    year = int(input('Enter year (yyyy) (-1 to quit): '))
    while (year < 1800 or year > 2099) and year != -1:
        year = int(input('INVALID - Enter year(1800-2099): '))

    if year == -1:
        terminate = True
    else:
        # determine if leap year
        if (year % 4 == 0) and (not (year % 100 == 0) or
            (year % 400 == 0)):
            leap_year = True
        else:
            leap_year = False

        # determine day of the week
        century_digits = year // 100
        year_digits = year % 100
        value = year_digits + (year_digits // 4)

        if century_digits == 18:
            value = value + 2
        elif century_digits == 20:
            value = value + 6

        # leap year check
        if not leap_year:
            value = value + 1

        # determine first day of month for Jan 1
        first_day_of_current_month = (value + 1) % 7

        # construct calendar for all 12 months
        for month_num in range(12):
            month_name = month_names[month_num]

            # init for new month
            current_day = 1
            if first_day_of_current_month == 0:
                starting_col = 7
            else:
                starting_col = first_day_of_current_month

```

```
            current_col = 1
            calendar_week = ''
            calendar_month = []

            # add any needed leading space for first week of month
            while current_col < starting_col:
                calendar_week = calendar_week + blank_col
                current_col = current_col + 1

            # store month as separate weeks
            if (month_name == 'February') and leap_year:
                num_days_this_month = 29
            else:
                num_days_this_month = days_in_month[month_num]

            while current_day <= num_days_this_month:

                # store day of month in field of length 3
                calendar_week = calendar_week + \
                   format(str(current_day),'>3')

                # check if at last column of displayed week
                if current_col == 7:
                    calendar_month = calendar_month + [calendar_week]
                    calendar_week = ''
                    current_col = 1
                else:
                    current_col = current_col + 1

                # increment current day
                current_day = current_day + 1

            # fill out final row of month with needed blanks
            calendar_week = calendar_week + \
                            blank_week[0:(7-current_col+1) * 3]
            calendar_month = calendar_month + [calendar_week]

            # reset values for next month
            first_day_of_current_month = current_col
            calendar_year = calendar_year + [calendar_month]
            calendar_month = []

        # print calendar year
        print('\n', year,'\n')

        # each row starts with January, April, July, or October
        for month_num in [0,3,6,9]:

            # displays three months in each row
            for i in range(month_num, month_num + 3):
                  print(' ' + format(month_names[i],'19'),
                        month_separator, end='')

            # display each week of months on separate lines
            week = 0
            lines_to_print = True
```

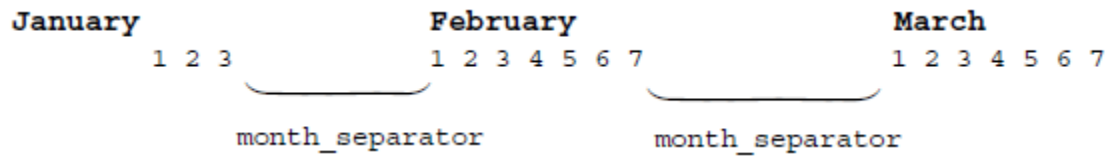
Notes:

In this final version, the only change at the start of the program is that a program greeting is added on line 19. The rest of the program is the same up to line 105, the point where the calendar year has been constructed. (The `print(calendar_year)` line and re-initialization of `calendar_year` to the empty list have been removed from the previous version, since they were only there for testing purposes.)

The new code in this version of the program is in lines 107–141, which displays the calendar year.

On line 108 the year is displayed. Because the months are displayed three across, as shown in Figure 4-16, the for loop on line 111 iterates variable `month_num` over the values `[0, 3, 6, 9]`. Thus, when `month_num` is `0`, months `0-2` (January - March) are displayed. When `month_num` is 3, months `3-5` (April - June) are displayed, and so forth.

The for loop at line 114 displays the month names for each row (for example, January, February, and March). Each is displayed left-justified in a field width of 19. A leading blank character is appended to the formatting string to align with the first column of numbers displayed for each month. The `print(. . ., end5'')` form of print is used, which prevents the cursor from moving to the next line. Thus, the months can be displayed side-by-side. Variable `month_separator` contains the appropriate number of blank spaces (initialized at the top of the program) to provide the required amount of padding between the months, as shown below,

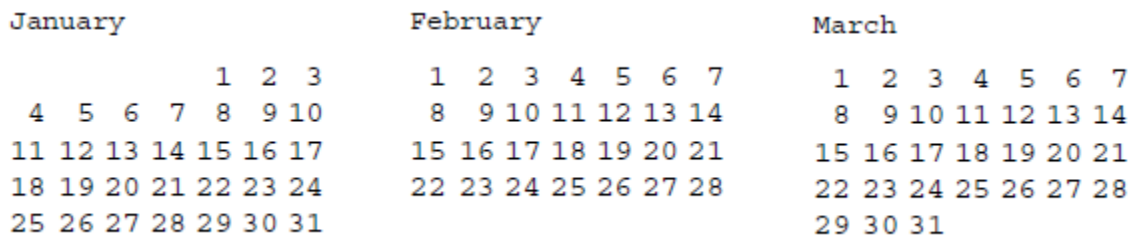


Lines 119–120 perform the initialization needed for the following while loop (at line 122), which displays each week, one-by-one, of the current three months. Variable `week` is initial- ized to zero for each month and is used to keep count of the number of weeks displayed. Variable `lines_to_print` is initialized to `True` to start the execution of the following while loop.

At line 125 within the while loop, `lines_to_print` is initialized to `False`. It is then set to `True` by any (or all) of the current three months being displayed only if they still have more calendar lines (weeks) to print, thus causing the while loop to continue with another iteration. This occurs within the for loop at lines 128–135. Since variable `month_num` indicates the current month being displayed, the number of weeks in the month is determined by the length of the tuple of strings for the current month `k`.

```
len(calendar_year[k])
```

Note that some months may have no more weeks to display, whereas others may. This is the case for the first three months of 2015,



In this case, the while loop needs to continue to iterate in order to display the last lines of January and March even though the last line of February has been displayed. Therefore, in cases where a given month has a line to print but another month doesn't, a blank line is displayed in order to maintain the correct alignment of month weeks. After the week of dates (or blank week) is output for each of the three months, the cursor is moved to the start of the next line (on line 138) and variable `week` is incremented by one (line 141) before the loop begins the next iteration for displaying the next row of calendar weeks.

Finally, the while loop at line 122 continues to iterate until there are no more lines to display for all of the three months currently being displayed—that is, until `lines_to_print` is `False`.

Final Testing

Complete the testing by executing the program on a set of test cases. Although the test plan is not as complete as it could be, it includes test cases for months from each century, including both leap years and non-leap years.

Calendar Month	Expected Results		Actual Results		Evaluation?
	First Day	num days	First Day	num days	
April 1912	Sunday	30			
February 1985	Monday	28			
May 2015	Tuesday	31			
January 1800	Wednesday	31			
February 1900	Thursday	28			
August 2031	Friday	29			
January 2011	Saturday	31			

If your test plan passes for all test cases, the output should look as follows: