When **line 50** is reached, Boolean flag `game_over` may be either `True` or `False`. It is `True` when the user has indicated that they have entered all their coin values (by hitting return), or if the total of the coin values entered exceeds the value in variable `amount`—it is `False` otherwise. Therefore, flag variable `game_over` is used to determine whether the user should be prompted to play another game (**line 51**). If they choose to quit the program when prompted, then Boolean variable `terminate` is set to `True`. This causes the encompassing while loop at **line 17** to terminate, leaving only the final "goodbye" message on **line 56** to be executed before the program terminates.

Self-Test Questions

1. A while loop continues to iterate until its condition becomes false. TRUE/FALSE
2. A while loop executes zero or more times. TRUE/FALSE
3. All iteration can be achieved by a while loop. TRUE/FALSE
4. An infinite loop is an iterative control structures that,
   (a) Loops forever and must be forced to terminate
   (b) Loops until the program terminates with a system error
   (c) Both of the above
5. The terms *definite loop* and *indefinite loop* are used to indicate whether,
   (a) A given loop executes at least once
   (b) The number of times that a loop is executed can be determined before the loop is executed.
   (c) Both of the above
6. A Boolean flag is,
   (a) A variable
   (b) Has the value `True` or `False`
   (c) Is used as a condition for control statements
   (d) All of the above

ANSWERS: 1. True, 2. True, 3. True, 4. (c), 5. (b), 6. (d)

COMPUTATIONAL PROBLEM SOLVING

3.5 Calendar Month Program

3.5.1 The Problem

The problem is to display a calendar month for any given month between January 1800 and December 2099. The format of the month should be as shown in Figure 3-23.

MAY 2012

Sun	Mon	Tues	Wed	Thur	Fri	Sat
		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		

FIGURE 3-23 Calendar Month Display

3.5.2 Problem Analysis

Two specific algorithms are needed for this problem. First, we need an algorithm for computing the first day of a given month for years 1800 through 2099. This algorithm is given in Chapter 1. The second needed algorithm is for appropriately displaying the calendar month, given the day of the week that

the first day falls on, and the number of days in the month. We shall develop this algorithm. The data representation issues for this problem are straight forward.

3.5.3 Program Design

Meeting the Program Requirements

We will develop and implement an algorithm that displays the month as given. There is no requirement of how the month and year are to be entered. We shall therefore request the user to enter the month and year as integer values, with appropriate input error checking.

Data Description

What needs to be represented in the program is the month and year entered, whether the year is a leap year or not, the number of days in the month, and which day the first of the month falls on. Given that information, the calendar month can be displayed. The year and month will be entered and stored as integer values, represented by variables `year` and `month`,

```
year = 2012     month = 5
```

The remaining values will be computed by the program based on the given year and month, as given below,

```
leap_year     num_days_in_month     day_of_week
```

Variable `leap_year` holds a Boolean (`True/False`) value. Variables `num_days_in_month` and `day_of_week` each hold integer values.

Algorithmic Approach

First, we need an algorithm for determining the day of the week that a given date falls on. The algorithm for this from Chapter 1 is reproduced in Figure 3-24.

We also need to determine how many days are in a given month, which relies on an algorithm for determining leap years for the month of February. The code for this has already been developed in the "Number of Days in Month" program in section 3.3.4. We shall also reuse the portion of code from that program for determining leap years, reproduced below.

```
if (year % 4 == 0) and (not (year % 100 == 0) or year % 400):
   leap_year = True
else:
   leap_Year = False
```

Let's review how this algorithm works, and try to determine the day of the week on which May 24, 2025 falls. First, variable `century_digits` (holding the first two digits of the year) is set to `20` and `year_digits` (holding the last two digits of the year) is set to `25` (**steps 1 and 2**). Variable `value`, in **step 3**, is then set to

```
value = year_digits + floor(year_digits / 4)
      = 25 + floor(25/4) → 25 + floor(6.25) → 25 + 6 → 31
```

To determine the day of the week for a given **month**, **day**, and **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 the **month** is equal to January and the **year** is not a leap year,
   then add 1 to **value**, else,

   if the **month** is equal to February and the **year** is a leap year, then
   add 3 to **value**; if not a leap year, then add 4 to **value,** else,

   if the **month** is equal to March or November, then add 4 to **value**, else,

   if the **month** is equal to April or July, then add 0 to **value**, else,

   if the **month** is equal to May, then add 2 to **value**, else,

   if the **month** is equal to June, then add 5 to **value**, else,

   if the **month** is equal to August, then add 3 to **value**, else,

   if the **month** is equal to October, then add 1 to **value**, else,

   if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else

   if **value** is equal to 2, day of the week is Monday; else

   if **value** is equal to 3, day of the week is Tuesday; else

   if **value** is equal to 4, day of the week is Wednesday; else

   if **value** is equal to 5, day of the week is Thursday; else

   if **value** is equal to 6, day of the week is Friday; else

   if **value** is equal to 0, day of the week is Saturday

FIGURE 3-24 Day of the Week Algorithm (from Chapter 1)

In **step 4**, since `century_digits` is equal to `20`, `value` is incremented by 6,

```
value = value + 6 → 31 + 6 → 37
```

In **step 5**, since the month is equal to May, `value` is incremented by 2,

```
value = value + 2 → 37 + 2 → 39
```

In **step 6**, `value` is updated based on the day of the month. Since we want to determine the day of the week for the 24th (of May), `value` is updated as follows,

```
value = (value + day of the month) mod 7
      = (39 + 24) mod 7
      = 63 mod 7
      = 0
```

Therefore, by **step 7** of the algorithm, the day of the week for May 24, 2025 is a Saturday. A table for the interpretation of the day of the week for the final computed value is given in Figure 3-25.

1	2	3	4	5	6	0
Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday

FIGURE 3-25 Interpretation of the Day of the Week Algorithm Results

Overall Program Steps

The overall steps in this program design are given in Figure 3-26.

3.5.4 Program Implementation and Testing

Stage 1—Determining the Number of Days in the Month/Leap Years

We develop and test the program in three stages. First, we implement and test the code that determines, for a given month and year, the number of days in the month and whether the year is a leap year or not, given in Figure 3-27.

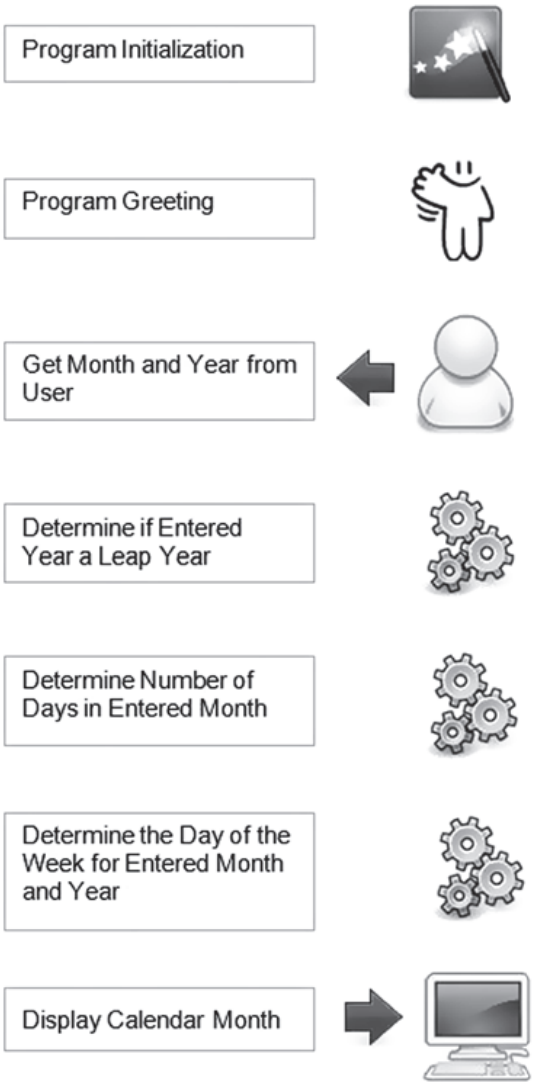

FIGURE 3-26 Overall Steps of Calendar Month Program

```
1   # Calendar Month Program (stage 1)
2
3   # init
4   terminate = False
5
6   # program greeting
7   print('This program will display a calendar month between 1800 and 2099')
8
9   while not terminate:
10      # get month and year
11      month = int(input('Enter month 1-12 (-1 to quit): '))
12
13      if month == -1:
14          terminate = True
15      else:
16          while month < 1 or month > 12:
17              month = int(input('INVALID INPUT - Enter month 1-12: '))
18
19          year = int(input('Enter year (yyyy): '))
20
21          while year < 1800 or year > 2099:
22              year = int(input('INVALID - Enter year (1800-2099): '))
23
24          # determine if leap year
25          if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
26              leap_year = True
27          else:
28              leap_year = False
29
30          # determine num of days in month
31          if month in (1, 3, 5, 7, 8, 10, 12):
32              num_days_in_month = 31
33          elif month in (4, 6, 9, 11):
34              num_days_in_month = 30
35          elif leap_year:  # February
36              num_days_in_month = 29
37          else:
38              num_days_in_month = 28
39
40          print ('\n', month, ',', year, 'has', num_days_in_month, 'days')
41
42          if leap_year:
43              print (year, 'is a leap year\n')
44          else:
45              print (year, 'is NOT a leap year\n')
```

FIGURE 3-27 First Stage of Calendar Month Program

The month and year entered by the user are stored in variables `month` and `year`. While loops are used at **lines 16** and **21** to perform input error checking. **Lines 25–28** are adapted from the previous Number of Days in Month program for determining leap years. **Lines 31–38** are similar to the previous program for determining the number of days in a month, stored in variable `num_days_in_month`. **Lines 42–45** contain added code for the purpose of testing. These instructions will not be part of the final program. The program continues to prompt for another month until −1 is entered. Thus, Boolean flag `terminate` is initialized to `False` (**line 4**) and set to `True` (**line 14**) when the program is to terminate.

Stage 1 Testing

We give output from the testing of this version of the program in Figure 3-28.

```
Enter month (1-12): 14
INVALID INPUT
Enter month (1-12): 1
Enter year (yyyy): 1800

1 , 1800 has 31 days
1800 is NOT a leap year
```

FIGURE 3-28 Example Output of First Stage Testing

The set of test cases for this stage of the program is given in Figure 3-29. The test cases are selected such that each month is tested within the 1800s, 1900s, and 2000s. The month of February has a number of test cases to ensure that the program is working for non-leap years (1985), "typical" leap years (1984), and exception years (1900 and 2000). The test plan also includes the "extreme" cases of January 1800 and December 2099 (the beginning and end of the range of valid months). All test cases are shown to have passed, and thus we can move on to stage 2 of the program development.

Calendar Month	Expected Results		Actual Results		Evaluation
	num days	leap year	num days	leap year	
January 1800	31	no	31	no	Passed
February 1900	28	no	28	no	Passed
February 1984	29	yes	29	yes	Passed
February 1985	28	no	28	no	Passed
February 2000	29	yes	29	yes	Passed
March 1810	31	no	31	no	Passed
April 1912	30	yes	30	yes	Passed
May 2015	31	no	31	no	Passed
June 1825	30	no	30	no	Passed
July 1928	31	yes	31	yes	Passed
August 2031	31	no	31	no	Passed
September 1845	30	no	30	no	Passed
October 1947	31	no	31	no	Passed
November 2053	30	no	30	no	Passed
December 2099	31	no	31	no	Passed

FIGURE 3-29 Results of Execution of Test Plan for Stage 1

Stage 2—Determining the Day of the Week

We give the next stage of the program in Figure 3-30. This version includes the code for determining the day of the week for the first day of a given month and year (**lines 40–71**), with the final print statement (**line 74**) displaying the test results. Note that for testing purposes, there is no need to convert the day number into the actual name (e.g., "Monday")—this "raw output" is good enough. Also, for this program, we will need to determine only the day of the week for the first day of any

```
# Calendar Month Program (stage 2)

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099'

while not terminate:
    # get month and year
    month = int(input('Enter month (1-12): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID INPUT - Enter month 1-12: '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

        # 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 num of days in month
        if month in (1,3,5,7,8,10,12):
            num_days_in_month = 31
        elif month in (4,6,9,11):
            num_days_in_month = 30
        elif leap_year:  # February
            num_days_in_month = 29
        else:
            num_days_in_month = 28

        # 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

```

FIGURE 3-30 Second Stage of Calendar Month Program (*Continued*)

given month, since all remaining days follow sequentially. Therefore, the day value in the day of the week algorithm part of the code is hard-coded to 1 (on **line 71**). Let's look at the code that implements the day of the week algorithm.

The algorithm operates separately on the first two digits and last two digits of the year. On **line 41**, integer division is used to extract the first two digits of the year (for example, `1860 // 100`

```
51          if month == 1 and not leap_year:
52              value = value + 1
53          elif month == 2:
54              if leap_year:
55                  value = value + 3
56              else:
57                  value = value + 4
58          elif month == 3 or month == 11:
59              value = value + 4
60          elif month == 5:
61              value = value + 2
62          elif month == 6:
63              value = value + 5
64          elif month == 8:
65              value = value + 3
66          elif month == 9 or month == 12:
67              value = value + 6
68          elif month == 10:
69              value = value + 1
70
71          day_of_week = (value + 1) % 7  # 1-Sunday, 2-Monday, ...,
72
73          # display results
74          print('Day of the week is', day_of_week)
```

FIGURE 3-30 Second Stage of Calendar Month Program

equals `18`). On **line 42**, the modulus operator, `%`, is used to extract the last two digits (for example, `1860 % 100` equals `60`). The rest of the program (through **line 71**) follows the day of the week algorithm given above.

Stage 2 Testing

We give a sample test run of this version of the program in Figure 3-31.

```
1
Enter month (1-12): 4
Enter year (yyyy): 1860
Day of the week is 1
Enter month (1-12): -1
>>>
```

FIGURE 3-31 Example Output of Second Stage Testing

Figure 3-32 shows the results of the execution of the test plan for this version of the program. It includes the same months as in the test plan for the first stage.

Since all test cases passed, we can move on to the final stage of program development.

Final Stage—Displaying the Calendar Month

In the final stage of the program (Figure 3-33), we add the code for displaying the calendar month.

The corresponding name for the month number is determined on **lines 74–97** and displayed (**line 100**). The while loop at **line 113** moves the cursor to the proper starting column by "printing"

Calendar Month	Expected Results first day of month	Actual Results first day of month	Evaluation
January 1800	4 (Wednesday)	4	Passed
February 1900	5 (Thursday)	5	Passed
February 1984	4 (Wednesday)	4	Passed
February 1985	6 (Friday)	6	Passed
February 2000	3 (Tuesday)	3	Passed
March 1810	5 (Thursday)	5	Passed
April 1912	2 (Monday)	2	Passed
May 2015	6 (Friday)	6	Passed
June 1825	4 (Wednesday)	4	Passed
July 1928	1 (Sunday)	1	Passed
August 2031	6 (Friday)	6	Passed
September 1845	2 (Monday)	2	Passed
October 1947	4 (Wednesday)	4	Passed
November 2053	0 (Saturday)	0	Passed
December 2099	3 (Tuesday)	3	Passed

FIGURE 3-32 Results of Execution of Test Plan for Stage 2

```
# Calendar Month Program

# init
terminate = False

# program greeting
print('This program will display a calendar month between 1800 and 2099')

while not terminate:
    # get month and year
    month = int(input('Enter month 1-12 (-1 to quit): '))

    if month == -1:
        terminate = True
    else:
        while month < 1 or month > 12:
            month = int(input('INVALID - Enter month (1-12): '))

        year = int(input('Enter year (yyyy): '))

        while year < 1800 or year > 2099:
            year = int(input('INVALID - Enter year (1800-2099): '))

```

FIGURE 3-33 Final Stage of Calendar Month Program (*Continued*)

```
24          # determine if leap year
25          if (year % 4 == 0) and (not(year % 100 == 0) or (year % 400 == 0)):
26              leap_year = True
27          else:
28              leap_year = False
29
30          # determine num of days in month
31          if month in (1,3,5,7,8,10,12):
32              num_days_in_month = 31
33          elif month in (4,6,9,11):
34              num_days_in_month = 30
35          elif leap_year:  # February
36              num_days_in_month = 29
37          else:
38              num_days_in_month = 28
39
40          # determine day of the week
41          century_digits = year // 100
42          year_digits = year % 100
43
44          value = year_digits + (year_digits // 4)
45
46          if century_digits == 18:
47              value = value + 2
48          elif century_digits == 20:
49              value = value + 6
50
51          if month == 1 and not leap_year:
52              value = value + 1
53          elif month == 2:
54              if leap_year:
55                  value = value + 3
56              else:
57                  value = value + 4
58          elif month == 3 or month == 11:
59              value = value + 4
60          elif month == 5:
61              value = value + 2
62          elif month == 6:
63              value = value + 5
64          elif month == 8:
65              value = value + 3
66          elif month == 9 or month == 12:
67              value = value + 6
68          elif month == 10:
69              value = value + 1
70
71          day_of_week = (value + 1) % 7  # 1-Sun, 2-Mon, ..., 0-Sat
72
```

FIGURE 3-33 Final Stage of Calendar Month Program (*Continued*)

the `column_width` number of blank characters (4) for each column to be skipped. The while loop at **line 119** displays the dates. Single-digit dates are output (**line 121**) with three leading spaces, and two-digit dates with two (**line 123**) so that the columns line up. Each uses the **newline suppression form of print**, `print(..., end='')` to prevent the cursor from moving to the next screen line until it is time to do so.

Variable `current_day` is incremented from `1` to the number of days in the month. Variable `current_col` is also incremented by `1` to keep track of what column the current date is being

```
        # determine month name
        if month == 1:
            month_name = 'January'
        elif month == 2:
            month_name = 'February'
        elif month == 3:
            month_name = 'March'
        elif month == 4:
            month_name = 'April'
        elif month == 5:
            month_name = 'May'
        elif month == 6:
            month_name = 'June'
        elif month == 7:
            month_name = 'July'
        elif month == 8:
            month_name = 'August'
        elif month == 9:
            month_name = 'September'
        elif month == 10:
            month_name = 'October'
        elif month == 11:
            month_name = 'November'
        else:
            month_name = 'December'

        # display month and year heading
        print('\n', ' ' + month_name, year)

        # display rows of dates
        if day_of_week == 0:
            starting_col = 7
        else:
            starting_col = day_of_week

        current_col = 1
        column_width = 4
        blank_char = ' '
        blank_column = format(blank_char, str(column_width))

        while current_col <= starting_col:
            print(blank_column, end='')
            current_col = current_col + 1

        current_day = 1

        while current_day <= num_days_in_month:
            if current_day < 10:
                print (format(blank_char, '3') + str(current_day), end='')
            else:
                print (format(blank_char, '2') + str(current_day), end='')

            if current_col <= 7:
                current_col = current_col + 1
            else:
                current_col = 1
                print()

            current_day = current_day + 1

        print('\n')
```

FIGURE 3-33 Final Stage of Calendar Month Program

displayed in. When `current_col` equals 7, it is reset to `1` (**line 128**) and `print()` moves the cursor to the start of the next line (**line 129**). Otherwise, `current_col` is simply incremented by 1 (**line 126**).

An example test run of this final version of the program is given in Figure 3-34.

```
This program will display a calendar month between 1800 and 2099

Enter month 1-12 (-1 to quit): 1
Enter year (yyyy): 1800

  January   1800
                    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 month (1-12): -1
>>>
```

FIGURE 3-34 Example Output of Final Stage Testing

Something is obviously wrong. The calendar month is displayed with eight columns instead of seven. The testing of all other months produces the same results. Since the first two stages of the program were successfully tested, the problem must be in the code added in the final stage. The code at **line 74** simply assigns the month name. Therefore, we reflect on the logic of the code starting on **line 103**.

Lines 128–129 is where the column is reset back to column 1 and a new screen line is started, based on the current value of variable `current_col`,

```
if current_col <= 7:
   current_col = current_col + 1
else:
   current_col = 1
   print()
```

Variable `current_col` is initialized to `1` at **line 108**, and is advanced to the proper starting column on **lines 113–115**. Variable `starting_col` is set to the value (`0-6`) for the day of the week for the particular month being displayed. Since the day of the week results have been successfully tested, we can assume that `current_col` will have a value between 0 and 6. With that assumption, we can step though **lines 125–129** and see if this is where the problem is. Stepping through a program on paper by tracking the values of variables is referred to as **deskchecking**. We check what happens as the value of `current_col` approaches 7, as shown in Figure 3-35.

Now it is clear what the problem is—the classic *"off by one" error*! The condition of the while loop should be `current_col < 7`, *not* `current_col <= 7`. `Current_col` should be reset to `1` once the seventh column has been displayed (when `current_col` is `7`). Using the

Current value of **current_col**	Value of condition **current_col <= 7**	Updated value of **current_col**
5	True	6
6	True	7
7	True	8
8	False	1
1	True	2
etc.		

FIGURE 3-35 Deskchecking the Value of Variable `current_col`

<= operator causes `current_col` to be reset to `1` only after an *eighth* column is displayed. Thus, we make this correction in the program,

```
if current_col < 7:
    current_col = current_col + 1
else:
    current_col = 1
print()
```

After re-executing the program with this correction we get the current output, depicted in Figure 3-36.

```
This program will display a calendar month between 1800 and 2099

Enter month 1-12 (-1 to quit): 1
Enter year (yyyy): 1800

  January   1800
                    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 month (1-12): -1
>>>
```

FIGURE 3-36 Display Output of Final Stage of Calendar Month Program

Although the column error has been corrected, we find that the first of the month appears under the wrong column—the month should start on a Wednesday (fourth column), not a Thursday column (fifth column). The problem must be in how the first row of the month is displayed. Other months are tested, each found to be off by one day. We therefore look at **lines 113–115** that are responsible for moving over the cursor to the correct starting column,

```
while current_col <= starting_col:
    print(blank_column, end='')
    current_col = current_col + 1
```

We consider whether there is another "off by one" error. Reconsidering the condition of the while loop, we realize that, in fact, this is the error. If the correct starting column is 4 (Wednesday), then the cursor should move past three columns and place a 1 in the fourth column. The current condition, however, would move the cursor past *four* columns, thus placing a 1 in the fifth column (Thursday). The corrected code is given below.

```
while current_col < starting_col:
    print(' ', end='')
    current_col = current_col + 1
```

The month is now correctly displayed. We complete the testing by executing the program on a set of test cases (Figure 3-37). 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 first day of month	num days	Evaluation
April 1912	Sunday	30	Passed
February 1985	Monday	28	Passed
May 2015	Tuesday	31	Passed
January 1800	Wednesday	31	Passed
February 1900	Thursday	28	Passed
February 1984	Friday	29	Passed
January 2011	Saturday	31	Passed

FIGURE 3-37 Results of Execution of Test Plan for Final Stage

CHAPTER SUMMARY

General Topics

Control Statement/Control Structure
Sequential, Selection, and Iterative Control
Relational Operators/Boolean Operators/
    Boolean Expressions
Operator Precedence and Boolean Expressions
Logically Equivalent Boolean Expressions
Short-Circuit (Lazy) Evaluation
Selection Control Statements/if Statement
Compound Statement
Multi-way Selection
While Statements
Input Error Checking
Infinite Loops
Definite vs. Indefinite Loops
Boolean Flags and Indefinite Loops
Deskchecking

Python-Specific Programming Topics

Membership Operators `in`, `not in`
`if` Statement in Python/`else` and `elif` headers
Indentation in Python
Multi-way Selection in Python
`while` Statement in Python