

Chapter 3 Control Structures

In Chapter 2 we looked at the “nuts and bolts” of programming. In this chapter, we discuss the three fundamental means of controlling the order of execution of instructions within a program, referred to as sequential, selection, and iterative control.

Motivation

The first electronic computers over sixty years ago were referred to as “Electronic Brains.” This gave the misleading impression that computers could “think.” Although very complex in their design, computers are machines that simply do, step-by-step (instruction-by-instruction), what they are told. Thus, there is no more intelligence in a computer than what it is instructed to do.

What computers can do, however, is to execute a series of instructions very quickly and very reliably. It is the speed in which instructions can be executed that gives computers their power, since the execution of many simple instructions can result in very complex behavior. And thus this is the enticement of computing. A computer can accomplish any task for which there is an algorithm for doing it. The instructions could be for something as simple as sorting lists, or as ambitious as performing intelligent tasks that as of now only humans are capable of performing.

In this chapter, we look at how to control the sequence of instructions that are executed in Python.

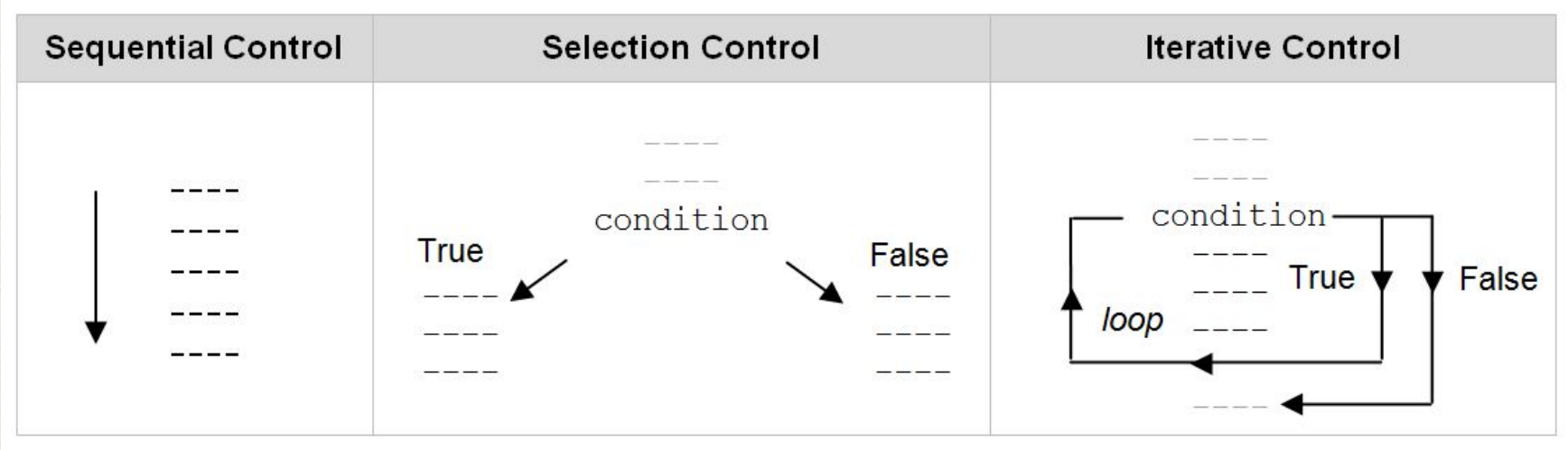
Term	Number of Floating Point Operations / Second		Device
Megaflops	10^6	1 million FLOPS	Supercomputers (1970s)
Gigaflops	10^9	1 billion FLOPS	CPU (single core)
Teraflops	10^{12}	1 trillion FLOPS	CPU (multi-core)
Petaflops	10^{15}	1 quadrillion FLOPS	Supercomputers (current)
Exaflops	10^{18}	1 quintillion FLOPS	Supercomputers in 2020 (projected)

What is a Control Structure?

Control flow is the **order that instructions are executed in a program**. A **control statement** is a statement that determines control flow of a set of instructions.

There are **three fundamental forms of control that programming languages provide**,

- **sequential control**
- **selection control**
- **iterative control**



Boolean Expressions

The **Boolean data type** contains two Boolean values, denoted as **True** and **False** in Python.

A **Boolean expression** is an **expression that evaluates to a Boolean value**. Boolean expressions are used to denote the conditions for selection and iterative control statements.

Relational Operators

The **relational operators** in Python **perform the usual comparison operations**.

Relational expressions are a type of **Boolean expression**, since they evaluate to a Boolean result.

Relational Operators in Python

Relational Operators	Example	Result
<code>==</code> equal	<code>10 == 10</code>	True
<code>!=</code> not equal	<code>10 != 10</code>	False
<code><</code> less than	<code>10 < 20</code>	True
<code>></code> greater than	<code>'Alan' > 'Brenda'</code>	False
<code><=</code> less than or equal to	<code>10 <= 10</code>	True
<code>>=</code> greater than or equal to	<code>'A' >= 'D'</code>	False

Note that these operators not only apply to numeric values, but to any set of values that has an ordering, such as strings.

Let's Try It

What do each of the following relational expressions evaluate to?

```
>>> 10 == 20  
???
```

```
>>> 10 != 20  
???
```

```
>>> 10 <= 20  
???
```

```
>>> '2' < '9'  
???
```

```
>>> '12' < '9'  
???
```

```
>>> '12' > '9'  
???
```

```
>>> 'Hello' == "Hello"  
???
```

```
>>> 'Hello' < 'Zebra'  
???
```

```
>>> 'hello' < 'ZEBRA'  
???
```

Membership Operators

Python provides a convenient pair of **membership operators**. These operators can be used to easily determine if a particular value occurs within a specified list of values.

Membership Operators	Examples	Result
<code>in</code>	<code>10 in (10, 20, 30)</code> <code>red in ('red', 'green', 'blue')</code>	True True
<code>not in</code>	<code>10 not in (10, 20, 30)</code>	False

The **membership operators** can also be used to check if a given string occurs within another string,

```
>>> 'Dr.' in 'Dr. Madison'  
True
```

As with the relational operators, the membership operators can be used to construct Boolean expressions.

Let's Try It

What do each of the following relational expressions evaluate to?

```
>>> 10 in (40, 20, 10)
???
```

```
>>> 10 not in (40, 20, 10)
???
```

```
>>> .25 in (.45, .25, .65)
???
```

```
>>> grade = 'A'
>>> grade in ('A', 'B', 'C', 'D', 'F')
???
```

```
>>> city = 'Houston'
>>> city in ('NY', 'Baltimore', 'LA')
???
```

Boolean Operators

George Boole, in the mid-1800s, developed what we now call **Boolean algebra**. His goal was to develop an algebra based on true/false rather than numerical values.

Boolean algebra contains a set of Boolean (logical) operators, denoted by **and**, **or**, and **not**. These logical operators can be used to construct arbitrarily complex Boolean expressions.

x	y	x and y	x or y	not x
False	False	False	False	True
True	False	False	True	False
False	True	False	True	True
True	True	True	True	False

Logical and is true only when *both* its operands are true — otherwise, it is false. **Logical or** is true when *either or both* of its operands are true, and thus false only when both operands are false. **Logical not** simply reverses truth values — not False equals True, and not True equals False.

Consider the following :

$$1 \leq \text{num} \leq 10$$

Although in mathematics this notation is understood, consider how this would be evaluated in a programming language (for num equal to 15):

$$1 \leq \text{num} \leq 10 \rightarrow 1 \leq 15 \leq 10 \rightarrow \text{True} \leq 10 \rightarrow \text{?!?}$$

The subexpression for the left relational operator would be evaluated first, which evaluates to True. Continuing, however, it doesn't make sense to check if True is less than or equal to 10. Some programming languages would generate a mixed-type expression error for this.

Therefore, the correct way for computer evaluation of the condition is by use of the Boolean **and** operator (again for num equal to 15):

$1 \leq \text{num} \text{ and } \text{num} \leq 10 \rightarrow (1 \leq \text{num}) \text{ and } (\text{num} \leq 10) \rightarrow$
 $\text{True and } (\text{num} \leq 10) \rightarrow \text{True and True} \rightarrow \text{True}$

Let's see what we get when we do evaluate the expression in the Python shell (for num equal to 15)

```
>>> 1 <= num and num <= 10
False
```

We actually get the correct result, False. If we were to try the original form of the expression,

```
>>> 1 <= num <= 10
False
```

This also works without error in Python?!

What you need to be aware of is that Python allows a relational expression of the form,

$$1 \leq \text{num} \leq 10$$

but Python automatically converts it to the form before evaluated,

$$1 \leq \text{num} \text{ and } \text{num} \leq 10$$

Thus, although Python offers this convenient shorthand, many programming languages require the longer form expression by use of logical and, and would give an error (or incorrect results) if written in the shorter form. Thus, as a novice programmer, *it would be best not to get in the habit of using the shorter form expression particular to Python.*

Let's Try It

What do each of the following relational expressions evaluate to?

```
>>> True and False  
???
```

```
>>> True or False  
???
```

```
>>> not(True) and False  
???
```

```
>>> not(True and False)  
???
```

```
>>> (10 < 0) and (10 > 2)  
???
```

```
>>> (10 < 0) or (10 > 2)  
???
```

```
>>> not(10 < 0) or (10 > 2)  
???
```

```
>>> not(10 < 0 or 10 > 2)  
???
```

Operator Precedence and Boolean Expressions

We saw the notion of operator precedence related to the use of arithmetic expressions.

Operator precedence applies to Boolean operators as well.

Operator Precedence and Boolean Expressions

Operator	Associativity
** (exponentiation)	right-to-left
- (negation)	left-to-right
* (mult), / (div), // (truncating div), % (modulo)	left-to-right
+ (addition), - (subtraction)	left-to-right
<, >, <=, >=, !=, == (relational operators)	left-to-right
not	left-to-right
and	left-to-right
or	left-to-right

As we saw earlier, in the table, **high-priority operators are placed before lower-priority operators**. Thus we see that all arithmetic operators are performed before any relational or Boolean operators.

Unary Boolean operator `not` has higher precedence than `and`, and Boolean operator `and` has higher precedence than the `or` operator.

Operator	Associativity
<code>**</code> (exponentiation)	right-to-left
<code>-</code> (negation)	left-to-right
<code>*</code> (mult), <code>/</code> (div), <code>//</code> (truncating div), <code>%</code> (modulo)	left-to-right
<code>+</code> (addition), <code>-</code> (subtraction)	left-to-right
<code><</code> , <code>></code> , <code><=</code> , <code>>=</code> , <code>!=</code> , <code>==</code> (relational operators)	left-to-right
<code>not</code>	left-to-right
<code>and</code>	left-to-right
<code>or</code>	left-to-right

As with arithmetic expressions, it is good programming practice for use parentheses, even if not needed, to add clarity and enhance readability,

```
(10 < 20 and 30 < 20) or (30 < 40)
(not 10 < 20) or (30 < 20)
```

If not all subexpressions,

```
((10 < 20) and (30 < 20)) or (30 < 40)
(not (10 < 20)) or (30 < 20)
```

Let's Try It

From the Python shell, enter the following and observe the results.

```
>>> not True and False  
???
```

```
>>> not True and False or True  
???
```

```
>>> 10 < 0 and not 10 > 2  
???
```

```
>>> not (10 < 0 or 10 < 20)  
???
```

Short-Circuit Evaluation

There are differences in how Boolean expressions are evaluated in different programming languages. For logical **and**, if the first operand evaluates to false, then regardless of the value of the second operand, the expression is false. Similarly, for logical **or**, if the first operand evaluates to true, regardless of the value of the second operand, the expression is true.

Because of this, some programming languages do not evaluate the second operand when the result is known by the first operand alone, called **short-circuit (lazy) evaluation**.

Subtle errors can result if the programmer is not aware of short-circuit evaluation. For example, the expression

```
if n != 0 and 1/n < tolerance:
```

would evaluate without error for all values of n when short-circuit evaluation is used. If programming in a language not using short-circuit evaluation, however, a “divide by zero” error would result when n is equal to 0. In such cases, the proper construction would be,

```
if n != 0:  
    if 1/n < tolerance:
```

In the Python programming language, short-circuit evaluation is used.

Logically Equivalent Boolean Expressions

In numerical algebra, there are arithmetically equivalent expressions of different form.

For example, $x(y + z)$ and $xy + xz$ are equivalent for any numerical values x , y , and z . Similarly, **there are logically equivalent Boolean expressions of different form.**

Logically equivalent Boolean expressions of different form.

(1) `(num != 0)`
`not (num == 0)`

... -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10

(2) `(num != 0) and (num != 6)`
`not (num == 0 or num == 6)`

... -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10...

(3) `(num >= 0) and (num <= 6)`
`(not num < 0) and (not num > 6)`
`not (num < 0 or num > 6)`

... -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10

(4) `(num < 0) or (num > 6)`
`(not num >= 0) and (not num <= 6)`
`not (num >= 0 or num <= 6)`

... -10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10

Fundamental logically equivalent Boolean expressions.

Logically Equivalent Boolean Expressions		
<code>x < y</code>	is equivalent to	<code>not (x >= y)</code>
<code>x <= y</code>	is equivalent to	<code>not (x > y)</code>
<code>x == y</code>	is equivalent to	<code>not (x != y)</code>
<code>x != y</code>	is equivalent to	<code>not (x == y)</code>
<code>not (x and y)</code>	is equivalent to	<code>(not x) or (not y)</code>
<code>not (x or y)</code>	is equivalent to	<code>(not x) and (not y)</code>

The last two equivalences are referred to as **De Morgan's Laws**.

Let's Try It

From the Python shell, enter the following and observe the results.

```
>>> 10 < 20
???
```

```
>>> not(10 >= 20)
???
```

```
>>> 10 != 20
???
```

```
>>> not (10 == 20)
???
```

```
>>> not(10 < 20 and 10 < 30)
???
```

```
>>> (not 10 < 20) or (not 10 < 30)
???
```

```
>>> not(10 < 20 or 10 < 30)
???
```

```
>>> (not 10 < 20) and (not 10 < 30)
???
```

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`
4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```
5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```
6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS:

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`
4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```
5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```
6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative,

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`
4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```
5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```
6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative 2. (a,b,d),

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`

4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```

5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```

6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a),

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`

4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```

5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```

6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a) 4. 16,

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`

4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```

5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```

6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a), 4. 16, 5. False,

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`
4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```
5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```
6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a), 4. 16, 5. False, 6. (a) True, (b) False,

Self-Test Questions

1. Three forms of control in programming are sequential, selection, and _____ control.
2. Which of the following expressions evaluate to True?
(a) `10 >= 8` (b) `8 <= 10` (c) `10 == 8` (d) `10 != 8` (e) `'8' < '10'`
3. Which of the following Boolean expressions evaluate to True?
(a) `'Dave' < 'Ed'` (b) `'dave' < 'Ed'` (c) `'Dave' < 'Dale'`
4. What is the value of variable `num` after the following is executed?

```
>>> num = 10
>>> num = num + 5
>>> num == 20
>>> num = num + 1
```
5. What does the following expression evaluate to for name equal to `'Ann'`?

```
name in ('Jacob', 'MaryAnn', 'Thomas')
```
6. Evaluate the following Boolean expressions using the operator precedence rules of Python.
(a) `10 >= 8 and 5 != 3` (b) `10 >= 8 and 5 == 3 or 14 < 5`
7. Which one of the following Boolean expressions is not logically equivalent to the other two?
(a) `not (num < 0 or num > 10)`
(b) `num > 0 and num < 10`
(c) `num >= 0 and num <= 10`

ANSWERS: 1. Iterative, 2. (a,b,d), 3. (a), 4. 16, 5. False, 6. (a) True, (b) False 7. (b)

Selection Control

A **selection control statement** is a **control statement** providing **selective execution of instructions**. A **selection control structure** is a given set of instructions and the selection control statement(s) controlling their execution. We look at the **if statement** providing selection control in Python next.

If Statement

An if statement is a selection control statement based on the value of a given Boolean expression.

if statement	Example use
<pre>if <i>condition</i>: <i>statements</i> else: <i>statements</i></pre>	<pre>if grade >= 70: print('passing grade') else: print('failing grade') if grade == 100: print('perfect score!')</pre>

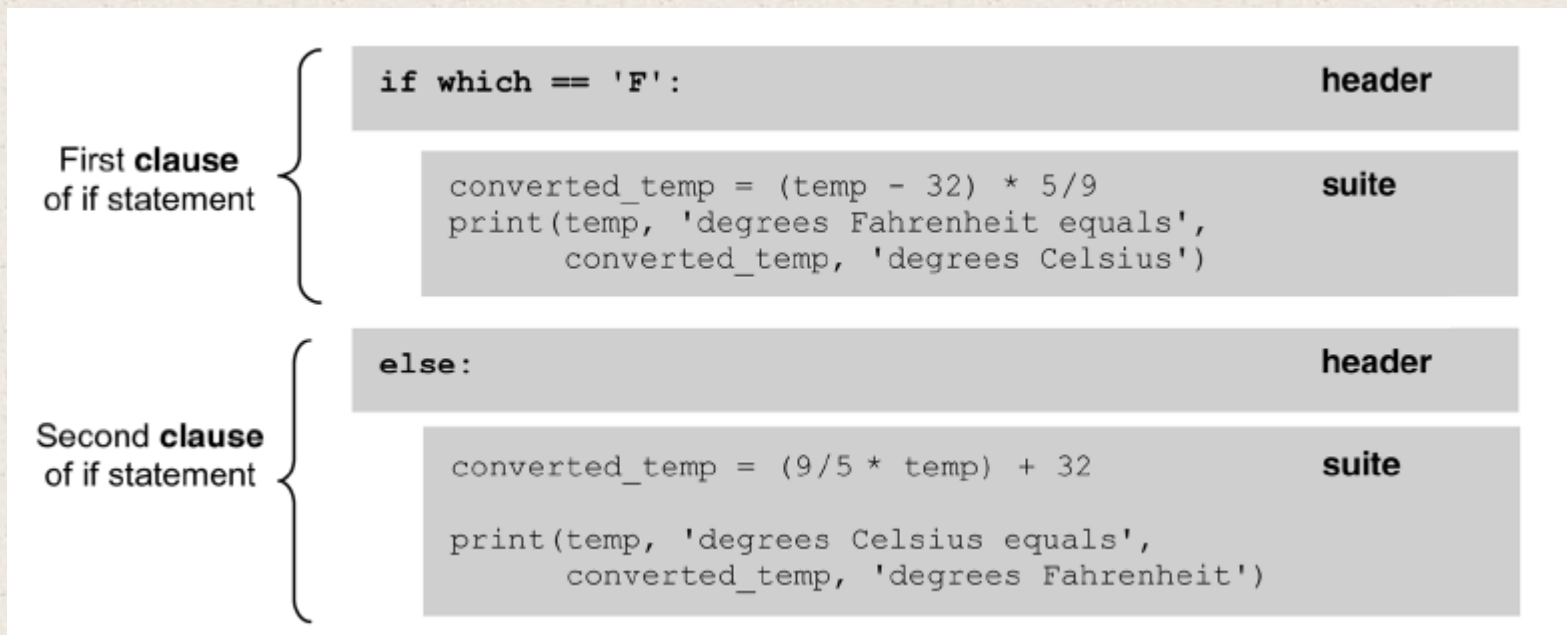
Note that if statements may omit the “else” part.

Below is a version of the temperature conversion program that allows the user to select a conversion of Fahrenheit to Celsius, or Celsius to Fahrenheit, implemented by the use of an if statement.

```
1 # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3 # Display program welcome
4 print('This program will convert temperatures (Fahrenheit/Celsius)')
5 print('Enter (F) to convert Fahrenheit to Celsius')
6 print('Enter (C) to convert Celsius to Fahrenheit')
7
8 # Get temperature to convert
9 which = input('Enter selection: ')
10 temp = int(input('Enter temperature to convert: '))
11
12 # Determine temperature conversion needed and display results
13 if which == 'F':
14     converted_temp = (temp - 32) * 5/9
15     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
16 else:
17     converted_temp = (9/5 * temp) + 32
18     print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
```

Indentation in Python

One fairly unique aspect of Python is that the amount of indentation of each program line is significant. In most programming languages, indentation has no affect on program logic—it is simply used to align program lines to aid readability. In Python, however, indentation is used to associate and group statements.



A **header** in Python is a specific keyword followed by a colon. In the example, the if-else statement contains two headers, “if which 5 = 'F':” containing keyword if, and “else:” consisting only of the keyword else. Headers that are part of the same compound statement must be indented the same amount—otherwise, a syntax error will result.

The set of statements following a header in Python is called a **suite** (commonly called a **block**). The statements of a given suite must all be indented the same amount. A header and its associated suite are together referred to as a **clause**.

A **compound statement** in Python may consist of one or more clauses. While four spaces is commonly used for each level of indentation, any number of spaces may be used, as shown below.

Valid indentation		Invalid indentation	
(a) <pre>if condition: statement statement else: statement statement</pre>	(b) <pre>if condition: statement statement else: statement statement</pre>	(c) <pre>if condition: statement statement else: statement statement</pre>	(d) <pre>if condition: statement statement else: statement statement</pre>

Let's Try It

From IDLE, create and run a Python program containing the code on the left and observe the results. Modify and run the code to match the version on the right and again observe the results. Make sure to indent the code exactly as shown.

```
grade = 90
if grade >= 70:
    print('passing grade')
else:
    print('failing grade')
```

```
grade = 90
if grade >= 70:
    print('passing grade')
else:
    print('failing grade')
```

Multi-Way Selection

Python provides two means of constructing multi-way selection — one involving **multiple nested if statements**, and the other involving a single if statement and the use of elif headers.

Multi-Way Selection by Use of Nested If Statements

Nested if statements

```
if condition:
    statements
else:
    if condition:
        statements
    else:
        if condition:
            statements

    etc.
```

Example use

```
if grade >= 90:
    print('Grade of A')
else:
    if grade >= 80:
        print('Grade of B')
    else:
        if grade >= 70:
            print('Grade of C')
        else:
            if grade >= 60:
                print('Grade of D')
            else:
                print('Grade of F')
```

Below is a version of the temperature conversion program that checks for invalid input, implemented by the use of nested if statements.

```
1 # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3 # Display program welcome
4 print('This program will convert temperatures (Fahrenheit/Celsius)')
5 print('Enter (F) to convert Fahrenheit to Celsius')
6 print('Enter (C) to convert Celsius to Fahrenheit')
7
8 # Get temperature to convert
9 which = input('Enter selection: ')
10 temp = int(input('Enter temperature to convert: '))
11
12 # Determine temperature conversion needed and display results
13 if which == 'F':
14     converted_temp = format((temp - 32) * 5.0/9.0, '.1f')
15     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
16 else:
17     if which == 'C':
18         converted_temp = format((9.0/5.0 * temp) + 32, '.1f')
19         print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')
20     else:
21         print('INVALID INPUT')
```

Let's Try It

From IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.

```
credits = 45
if credits >= 90:
    print('Senior')
else:
    if credits >= 60:
        print('Junior')
    else:
        if credits >= 30:
            print('Sophomore')
        else:
            if credits >= 1:
                print('Freshman')
            else:
                print('* No Earned Credits *')
```

Multi-Way Selection by Use of elif Header

```
if grade >= 90:  
    print('Grade of A')  
elif grade >= 80:  
    print('Grade of B')  
elif grade >= 70:  
    print('Grade of C')  
elif grade >= 60:  
    print('Grade of D')  
else:  
    print('Grade of F')
```

Let's Try It

From IDLE, create and run a Python program containing the code below and observe the results. Make sure to indent the code exactly as shown.

```
credits = 45

if credits >= 90:
    print('Senior')
elif credits >= 60:
    print('Junior')
elif grade >= 30:
    print('Sophomore')
elif grade >= 1:
    print('Freshman')
else:
    print('* No Earned Credits *')
```

Let's Apply It

Number of Days in Month Program

The following Python program prompts the user for a given month (and year for February), and displays how many days are in the month. This program utilizes the following programming features:

► **if statement**

► **elif header**

Program Execution ...

This program will determine the number of days in a given month

```
Enter the month (1-12): 14
* Invalid Value Entered - 14 '*'
>>>
```

This program will determine the number of days in a given month

```
Enter the month (1-12): 2
Please enter the year (e.g., 2010): 2000
There are 29 days in the month
```

```

1 # Number of Days in Month Program
2
3 # program greeting
4 print('This program will display the number of days in a given month\n')
5
6 # init
7 valid_input = True
8
9 # get user input
10 month = int(input('Enter the month (1-12): '))
11
12 # determine num of days in month
13
14 # february
15 if month == 2:
16     year = int(input('Please enter the year (e.g., 2010): '))
17
18     if (year % 4 == 0) and (not (year % 100 == 0) or (year % 400 == 0)):
19         num_days = 29
20     else:
21         num_days = 28
22
23 # january, march, may, july, august, october, december
24 elif month in (1, 3, 5, 7, 8, 10, 12):
25     num_days = 31
26
27 # april, june, september, november
28 elif month in (4, 6, 9, 11):
29     num_days = 30
30
31 # invalid input
32 else:
33     print('* Invalid Value Entered - ', month, '*')
34     valid_input = False
35
36 # output result
37 if valid_input:
38     print('There are', num_days, 'days in the month')

```

On **line 7**, variable `valid_input` is initialized to `True` for input error-checking. **Line 10** prompts the user for the month (1–12), stored in variable `month`. On **line 15** the month of February is checked for.

Line 24 checks if `month` is equal to 1, 3, 5, 7, 8, 10, or 12. If true, then `num_days` is assigned to 31. If not true, **line 28** checks if `month` is equal to 4, 6, 9, or 11. If true, `num_days` is assigned to 30. If not true, then an invalid month value was entered, and `valid_input` is set to `False`.

Finally, the number of days in the month is displayed only if the input is valid (**line 38**).

Self-Test Questions

1. All if statements must contain either an `else` or `elif` header. (TRUE/FALSE)
2. A compound statement is,
 - (a) A statement that spans more than one line
 - (b) A statement that contains other statements
 - (c) A statement that contains at least one arithmetic expression
3. Which of the following statements are true regarding headers in Python?
 - (a) Headers begin with a keyword and end with a colon.
 - (b) Headers always occur in pairs.
 - (c) All headers of the same compound statement must be indented the same amount.
4. Which of the following statements is true?
 - (a) Statements within a suite can be indented a different amount.
 - (b) Statements within a suite can be indented a different amount as long as all headers in the statement that it occurs in are indented the same amount.
 - (c) All headers must be indented the same amount as all other headers in the same statement, and all statements in a given suite must be indented the same amount.
5. The `elif` header allows for,
 - (a) Multi-way selection that cannot be accomplished otherwise
 - (b) Multi-way selection as a single if statement
 - (c) The use of a “catch-all” case in multi-way selection

ANSWERS:

Self-Test Questions

1. All if statements must contain either an `else` or `elif` header. (TRUE/FALSE)
2. A compound statement is,
 - (a) A statement that spans more than one line
 - (b) A statement that contains other statements
 - (c) A statement that contains at least one arithmetic expression
3. Which of the following statements are true regarding headers in Python?
 - (a) Headers begin with a keyword and end with a colon.
 - (b) Headers always occur in pairs.
 - (c) All headers of the same compound statement must be indented the same amount.
4. Which of the following statements is true?
 - (a) Statements within a suite can be indented a different amount.
 - (b) Statements within a suite can be indented a different amount as long as all headers in the statement that it occurs in are indented the same amount.
 - (c) All headers must be indented the same amount as all other headers in the same statement, and all statements in a given suite must be indented the same amount.
5. The `elif` header allows for,
 - (a) Multi-way selection that cannot be accomplished otherwise
 - (b) Multi-way selection as a single if statement
 - (c) The use of a “catch-all” case in multi-way selection

ANSWERS: 1. False,

Self-Test Questions

1. All if statements must contain either an `else` or `elif` header. (TRUE/FALSE)
2. A compound statement is,
 - (a) A statement that spans more than one line
 - (b) A statement that contains other statements
 - (c) A statement that contains at least one arithmetic expression
3. Which of the following statements are true regarding headers in Python?
 - (a) Headers begin with a keyword and end with a colon.
 - (b) Headers always occur in pairs.
 - (c) All headers of the same compound statement must be indented the same amount.
4. Which of the following statements is true?
 - (a) Statements within a suite can be indented a different amount.
 - (b) Statements within a suite can be indented a different amount as long as all headers in the statement that it occurs in are indented the same amount.
 - (c) All headers must be indented the same amount as all other headers in the same statement, and all statements in a given suite must be indented the same amount.
5. The `elif` header allows for,
 - (a) Multi-way selection that cannot be accomplished otherwise
 - (b) Multi-way selection as a single if statement
 - (c) The use of a “catch-all” case in multi-way selection

ANSWERS: 1. False 2. (b),

Self-Test Questions

1. All if statements must contain either an `else` or `elif` header. (TRUE/FALSE)
2. A compound statement is,
 - (a) A statement that spans more than one line
 - (b) A statement that contains other statements
 - (c) A statement that contains at least one arithmetic expression
3. Which of the following statements are true regarding headers in Python?
 - (a) Headers begin with a keyword and end with a colon.
 - (b) Headers always occur in pairs.
 - (c) All headers of the same compound statement must be indented the same amount.
4. Which of the following statements is true?
 - (a) Statements within a suite can be indented a different amount.
 - (b) Statements within a suite can be indented a different amount as long as all headers in the statement that it occurs in are indented the same amount.
 - (c) All headers must be indented the same amount as all other headers in the same statement, and all statements in a given suite must be indented the same amount.
5. The `elif` header allows for,
 - (a) Multi-way selection that cannot be accomplished otherwise
 - (b) Multi-way selection as a single if statement
 - (c) The use of a “catch-all” case in multi-way selection

ANSWERS: 1. False, 2. (b) 3. (a) (c)

Self-Test Questions

1. All if statements must contain either an `else` or `elif` header. (TRUE/FALSE)
2. A compound statement is,
 - (a) A statement that spans more than one line
 - (b) A statement that contains other statements
 - (c) A statement that contains at least one arithmetic expression
3. Which of the following statements are true regarding headers in Python?
 - (a) Headers begin with a keyword and end with a colon.
 - (b) Headers always occur in pairs.
 - (c) All headers of the same compound statement must be indented the same amount.
4. Which of the following statements is true?
 - (a) Statements within a suite can be indented a different amount.
 - (b) Statements within a suite can be indented a different amount as long as all headers in the statement that it occurs in are indented the same amount.
 - (c) All headers must be indented the same amount as all other headers in the same statement, and all statements in a given suite must be indented the same amount.
5. The `elif` header allows for,
 - (a) Multi-way selection that cannot be accomplished otherwise
 - (b) Multi-way selection as a single if statement
 - (c) The use of a “catch-all” case in multi-way selection

ANSWERS: 1. False, 2. (b), 3. (a) (c) 4. (c)

Self-Test Questions

1. All if statements must contain either an `else` or `elif` header. (TRUE/FALSE)
2. A compound statement is,
 - (a) A statement that spans more than one line
 - (b) A statement that contains other statements
 - (c) A statement that contains at least one arithmetic expression
3. Which of the following statements are true regarding headers in Python?
 - (a) Headers begin with a keyword and end with a colon.
 - (b) Headers always occur in pairs.
 - (c) All headers of the same compound statement must be indented the same amount.
4. Which of the following statements is true?
 - (a) Statements within a suite can be indented a different amount.
 - (b) Statements within a suite can be indented a different amount as long as all headers in the statement that it occurs in are indented the same amount.
 - (c) All headers must be indented the same amount as all other headers in the same statement, and all statements in a given suite must be indented the same amount.
5. The `elif` header allows for,
 - (a) Multi-way selection that cannot be accomplished otherwise
 - (b) Multi-way selection as a single if statement
 - (c) The use of a “catch-all” case in multi-way selection

ANSWERS: 1. False, 2. (b), 3. (a) (c), 4. (c), 5. (b)

Iterative Control

An **iterative control statement** is a control statement providing the repeated execution of a set of instructions. An **iterative control structure** is a set of instructions and the iterative control statement(s) controlling their execution. Because of their repeated execution, iterative control structures are commonly referred to as “loops.” We look at one specific iterative control statement next, the while statement.

While Statement

A **while statement** is an iterative control statement that repeatedly executes a set of statements based on a provided Boolean expression (condition). All iterative control needed in a program can be achieved by use of the while statement.

while statement

Example use

```
while condition:  
    suite
```

```
sum = 0  
current = 1  
  
n = int(input('Enter value: '))  
  
while current <= n:  
    sum = sum + current  
    current = current + 1
```

As long as the condition of a while statement is true, the statements within the loop are (re)executed. Once the condition becomes false, the iteration terminates and control continues with the first statement after the while loop.

Note that it is possible that the first time a loop is reached, the condition may be false, and therefore the loop would never be executed.

Execution steps for adding the first three integers by use of the previous while loop.

Iteration	sum	current	current <= 3	sum = sum + current	current = current + 1
1	0	1	True	sum = 0 + 1 (1)	current = 1 + 1 (2)
2	1	2	True	sum = 1 + 2 (3)	current = 2 + 1 (3)
3	3	3	True	sum = 3 + 3 (6)	current = 3 + 1 (4)
4	6	4	False	loop termination	

Input Error Checking

The while statement is well suited for input error checking in a program. This is demonstrated in the revised version of the temperature conversion program.

```

1 # Temperature Conversion Program (Celsius-Fahrenheit / Fahrenheit-Celsius)
2
3 # Display program welcome
4 print('This program will convert temperatures (Fahrenheit/Celsius)')
5 print('Enter (F) to convert Fahrenheit to Celsius')
6 print('Enter (C) to convert Celsius to Fahrenheit')
7
8 # Get temperature to convert
9 which = input('Enter selection: ')
10
11 while which != 'F' and which != 'C':
12     which = input("Please enter 'F' or 'C': ")
13
14 temp = int(input('Enter temperature to convert: '))
15
16 # Determine temperature conversion needed and display results
17 if which == 'F':
18     converted_temp = format((temp - 32) * 5/9, '.1f')
19     print(temp, 'degrees Fahrenheit equals', converted_temp, 'degrees Celsius')
20 else:
21     converted_temp = format((9/5 * temp) + 32, '.1f')
22     print(temp, 'degrees Celsius equals', converted_temp, 'degrees Fahrenheit')

```

The while loop is used to force the user to re-enter if neither an 'F' (for conversion to Fahrenheit) or a 'C' (for conversion to Celsius) is not entered.

Let's Try It

In IDLE, create and run a simple program containing the code below and observe the results. Make sure to indent the code exactly as shown.

```
n = 10
sum = 0
current = 1

while current <= n:
    sum = sum + current
    current = current + 1

print(sum)
???
```

```
n = 10
sum = 0
current = 1

while current <= n:
    sum = sum + current
    current = current + 1

print(sum)
???
```

Infinite Loops

An **infinite loop** is an **iterative control structure** that **never terminates** (or eventually terminates with a system error). Infinite loops are generally the result of programming errors. For example, if the condition of a while loop can never be false, an infinite loop will result when executed.

Following is an example program containing an infinite loop.

```
# add up first n integers
sum = 0
current = 1

n = int(input('Enter value: '))

while current <= n:
    sum = sum + current
```

The while loop is an infinite loop (for any value of n 1 or greater) because the value of current is not incremented

Let's Try It

From IDLE, create and run a simple program containing the code below and observe the results. Indent the code exactly as shown. To terminate an executing loop, hit ctrl-C.

```
while True:
    print ('Looping')
???
```

```
n = 10
sum = 0
current = 1

while current <= n:
    sum = sum + current

print (sum)

???
```

```
n = 10
sum = 0
current = 1

while current <= n:
    sum = sum + current
    n = n - 1

print (sum)

???
```

Definite vs. Indefinite Loops

A **definite loop** is a program loop in which the number of times the loop will iterate can be determined before the loop is executed. Following is an example of a definite loop.

```
sum = 0
current = 1
n = input('Enter value: ')
while current <= n:
    sum = sum + current
    current = current + 1
```

Although it is not known what the value of n will be until the input statement is executed, its value *is known by the time the while loop is reached*. Thus, it will execute “ n times.”

An **indefinite loop** is a program loop in which the number of times that the loop will iterate cannot be determined before the loop is executed.

```
which = input("Enter selection: ")
while which != 'F' and which != 'C':
    which = input("Please enter 'F' or 'C': ")
```

In this case, the number of times that the loop will be executed depends on how many times the user mistypes the input.

Boolean Flags and Indefinite Loops

Often the condition of a given while loop is denoted by a single Boolean variable, called a **Boolean flag**.

Boolean variable **valid_entries** on **line 12** in the following program is an example of the use of a Boolean flag for controlling a while loop.

```
1 # Oil Change Notification Program
2
3 # display program welcome
4 print('This program will determine if your car is in need of an oil change')
5
6 # init
7 miles_between_oil_change = 7500 # num miles between oil changes
8 miles_warning = 500 # how soon to warn of needed oil change
9 valid_entries = False
10
11 # get mileage of last oil change and current mileage and display
12 while not valid_entries:
13     mileage_last_oilchange = int(input('Enter mileage of last oil change: '))
14     current_mileage = int(input('Enter current mileage: '))
15
16     if current_mileage < mileage_last_oilchange:
17         print('Invalid entry - current mileage entered is less than')
18         print('mileage entered of last oil change')
19     else:
20         miles_traveled = current_mileage - mileage_last_oilchange
21         valid_entries = True
22
23 if miles_traveled >= miles_oil_change:
24     print('You are due for an oil change')
25 elif miles_traveled >= miles_oil_change - miles_warning:
26     print('You will soon be due for an oil change')
27 else:
28     print('You are not in immediate need of an oil change')
```

Let's Apply It

Coin Change Exercise Program

The following program implements an exercise for children learning to count change. It displays a random value between 1 and 99 cents, and asks the user to enter a set of coins that sums exactly to the amount shown. The program utilizes the following programming features:

- ▶ while loop
- ▶ if statement
- ▶ Boolean flag
- ▶ random number generator

Program Execution ...

The purpose of this exercise is to enter a number of coin values that add up to a displayed target value.

Enter coins values as 1-penny, 5-nickel, 10-dime and 25-quarter.
Hit return after the last entered coin value.

Enter coins that add up to 63 cents, one per line.

```
Enter first coin: 25
Enter next coin: 25
Enter next coin: 10
Enter next coin:
Sorry - you only entered 60 cents.
```

```
Try again (y/n)?: y
Enter coins that add up to 21 cents, one per line.
```

```
Enter first coin: 11
Invalid entry
Enter next coin: 10
Enter next coin: 10
Enter next coin: 5
Sorry - total amount exceeds 21 cents.
```

```
Try again (y/n)?: y
Enter coins that add up to 83 cents, one per line.
```

```
Enter first coin: 25
Enter next coin: 25
Enter next coin: 25
Enter next coin: 5
Enter next coin: 1
Enter next coin: 1
Enter next coin: 1
Enter next coin:
Correct!
```

```
Try again (y/n)?: n
Thanks for playing ... goodbye
```

```

1 # Coin Change Exercise Program
2
3 import random
4
5 # program greeting
6 print('The purpose of this exercise is to enter a number of coin values')
7 print('that add up to to a displayed target value.\n')
8 print('Enter coins values as 1-penny, 5-nickel, 10-dime and 25-quarter')
9 print("Hit return after the last entered coin value.")
10 print('-----')
11
12 # init
13 terminate = False
14 empty_str = ''
15
16 # start game
17 while not terminate:
18     amount = random.randint(1,99)
19     print('Enter coins that add up to', amount, 'cents, one per line.\n')
20     game_over = False
21     total = 0
22
23     while not game_over:
24         valid_entry = False
25
26         while not valid_entry:
27             if total == 0:
28                 entry = input('Enter first coin: ')
29             else:
30                 entry = input('Enter next coin: ')
31
32             if entry in (empty_str, '1', '5', '10', '25'):
33                 valid_entry = True
34             else:
35                 print('Invalid entry')
36
37             if entry == empty_str:
38                 if total == amount:
39                     print('Correct!')
40                 else:
41                     print('Sorry - you only entered', total, 'cents.')
42
43             game_over = True
44         else:
45             total = total + int(entry)
46             if total > amount:
47                 print('Sorry - total amount exceeds', amount, 'cents.')
48                 game_over = True
49
50         if game_over:
51             entry = input('\nTry again (y/n)? : ')
52
53             if entry == 'n':
54                 terminate = True
55
56 print('Thanks for playing ... goodbye')

```

On **line 18**, function `randint` (imported on **line 3**) is called to randomly generate a coin value for the user to match, stored in variable `amount`. On **line 13** Boolean variable `terminate` is initialized to `False`, used to determine when the program terminates.

The game begins on **line 17**. The while loop is re-executed for each new game played (while `terminate` is `False`). The while loop on **line 23** is re-executed while the current game is still in play (while `game_over` is `false`). The game ends when either the user enters a blank line, in which case the result is displayed and `game_over` is set to `True` (**lines 37–43**), or if the total amount exceeds the amount to be matched On **line 26**, a third Boolean flag is used, `valid_entry`, for controlling whether the user should be prompted again because of invalid input (on **lines 45–48**).

Note the use of membership operator `in` (**line 32**).

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:

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,

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,

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. (c), 6. (d)

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),

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),

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)

Calendar Month Program

We look at the problem of displaying a given calendar month.

Calendar Month

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.

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		

Calendar Month

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.

Calendar Month 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.

Calendar Month Program Design

Data Description

What needs to be represented is the month and year entered, whether the year is a leap year, 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.

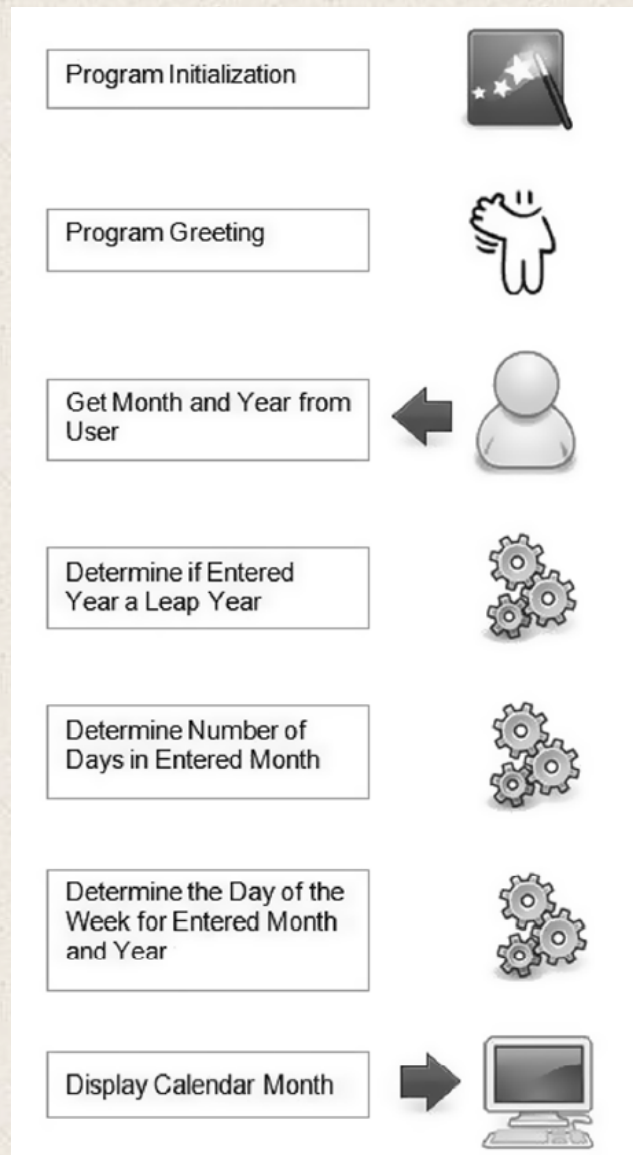
Algorithmic Approach

Algorithm for computing the day of the week. Have an algorithm for this.

Recall that the result of this algorithm is a value between 0-6 indicating the day of the week that a given date falls on.

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

For this program, we need only determine the day of the week for the first of the month. All the following days follow, for the given number of days in the month.



The Overall Steps of the Program

Calendar Month

Program Implementation

Stage 1—Input Validation / Determining Leap Years and Number of Days in Month

- **Prompts user for month and year.** Validates that value entered for month correct (1-12), and that year entered is in the range 1800-2099, inclusive.
- **Determines whether the year is a leap year.** Also determines the number of days in the month.

```

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')

```

Stage 1 Testing

We add test statements that displays then number of days in the month, and whether the year is a leap year or not. The test run below indicates that these values are correct of the given month and year.

```
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
```

A Set of Test Cases

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

Program Implementation

Stage 2—Determining the Day of the Week

Stage 2 of the program includes the code for determining the day of the week for the first day of a given month and year, with a final print statement 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.

```

1 # Calendar Month Program (stage 2)
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): '))
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

```

```

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)

```

Stage 2 Testing

Following is the output of a test run of the program. The day of the week value displayed is 1 (Sunday) which is the correct day of the week for the date entered.

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

A Set of Test Cases for Stage 2 Testing

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

Program Implementation

Final Stage – Displaying the Calendar Month

In the final stage of the program we add the code for displaying a calendar month.

```

1 # Calendar Month Program
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 - 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     # 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

```

```

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

```

```
73     # determine month name
74     if month == 1:
75         month_name = 'January'
76     elif month == 2:
77         month_name = 'February'
78     elif month == 3:
79         month_name = 'March'
80     elif month == 4:
81         month_name = 'April'
82     elif month == 5:
83         month_name = 'May'
84     elif month == 6:
85         month_name = 'June'
86     elif month == 7:
87         month_name = 'July'
88     elif month == 8:
89         month_name = 'August'
90     elif month == 9:
91         month_name = 'September'
92     elif month == 10:
93         month_name = 'October'
94     elif month == 11:
95         month_name = 'November'
96     else:
97         month_name = 'December'
98
```

```

99     # display month and year heading
100    print('\n', ' ' + month_name, year)
101
102    # display rows of dates
103    if day_of_week == 0:
104        starting_col = 7
105    else:
106        starting_col = day_of_week
107
108    current_col = 1
109    column_width = 4
110    blank_char = ' '
111    blank_column = format(blank_char, str(column_width))
112
113    while current_col <= starting_col:
114        print(blank_column, end='')
115        current_col = current_col + 1
116
117    current_day = 1
118
119    while current_day <= num_days_in_month:
120        if current_day < 10:
121            print (format(blank_char, '3') + str(current_day), end='')
122        else:
123            print (format(blank_char, '2') + str(current_day), end='')
124
125        if current_col <= 7:
126            current_col = current_col + 1
127        else:
128            current_col = 1
129            print()
130
131        current_day = current_day + 1
132
133    print('\n')

```

Stage 2 Testing

Following is the output of a test run of the program.

```
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
>>>
```

The month displayed is obviously incorrect, since each week is displayed with eight days! 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.

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`,

```
102     # display rows of dates
103     if day_of_week == 0:
104         starting_col = 7
105     else:
106         starting_col = day_of_week
107
108     current_col = 1
109     column_width = 4
110     blank_char = ' '
111     blank_column = format(blank_char, str(column_width))
112
113     while current_col <= starting_col:
114         print(blank_column, end='')
115         current_col = current_col + 1
116
117     current_day = 1
118
119     while current_day <= num_days_in_month:
120         if current_day < 10:
121             print (format(blank_char, '3') + str(current_day), end='')
122         else:
123             print (format(blank_char, '2') + str(current_day), end='')
124
125         if current_col <= 7:
126             current_col = current_col + 1
127         else:
128             current_col = 1
129             print()
130
131         current_day = current_day + 1
132
133     print('\n')
```

Variable `starting_col` is set to the value (0-6) for the day of the week for the particular month being displayed. Variable `current_col` is initialized to 1 at line 108 , and is advanced to the proper starting column on lines 113–115 .

```
102     # display rows of dates
103     if day_of_week == 0:
104         starting_col = 7
105     else:
106         starting_col = day_of_week
107
108     current_col = 1
109     column_width = 4
110     blank_char = ' '
111     blank_column = format(blank_char, str(column_width))
112
113     while current_col <= starting_col:
114         print(blank_column, end='')
115         current_col = current_col + 1
116
117     current_day = 1
118
119     while current_day <= num_days_in_month:
120         if current_day < 10:
121             print (format(blank_char, '3') + str(current_day), end='')
122         else:
123             print (format(blank_char, '2') + str(current_day), end='')
124
125         if current_col <= 7:
126             current_col = current_col + 1
127         else:
128             current_col = 1
129             print()
130
131         current_day = current_day + 1
132
133     print('\n')
```

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 through lines 125–129 and see if this is where the problem is.**

```
102     # display rows of dates
103     if day_of_week == 0:
104         starting_col = 7
105     else:
106         starting_col = day_of_week
107
108     current_col = 1
109     column_width = 4
110     blank_char = ' '
111     blank_column = format(blank_char, str(column_width))
112
113     while current_col <= starting_col:
114         print(blank_column, end='')
115         current_col = current_col + 1
116
117     current_day = 1
118
119     while current_day <= num_days_in_month:
120         if current_day < 10:
121             print (format(blank_char, '3') + str(current_day), end='')
122         else:
123             print (format(blank_char, '2') + str(current_day), end='')
124
125         if current_col <= 7:
126             current_col = current_col + 1
127         else:
128             current_col = 1
129             print()
130
131         current_day = current_day + 1
132
133     print('\n')
```

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 through lines 125–129 and see if this is where the problem is.**

```
102     # display rows of dates
103     if day_of_week == 0:
104         starting_col = 7
105     else:
106         starting_col = day_of_week
107
108     current_col = 1
109     column_width = 4
110     blank_char = ' '
111     blank_column = format(blank_char, str(column_width))
112
113     while current_col <= starting_col:
114         print(blank_column, end='')
115         current_col = current_col + 1
116
117     current_day = 1
118
119     while current_day <= num_days_in_month:
120         if current_day < 10:
121             print (format(blank_char, '3') + str(current_day), end='')
122         else:
123             print (format(blank_char, '2') + str(current_day), end='')
124
125         if current_col <= 7:
126             current_col = current_col + 1
127         else:
128             current_col = 1
129             print()
130
131         current_day = current_day + 1
132
133     print('\n')
```

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()
```

Deskchecking the Program Segment

We check what happens as the value of `current_col` approaches 7. Stepping through a program on paper is referred to as **deskchecking**.

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

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 `<=` operator causes `current_col` to be reset to 1 only *after* an eighth column is displayed.

After re-executing the program with this correction we get the following output.

```
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
>>>
```

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).

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**

```
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.

Calendar Month	Expected Results		Evaluation
	first day of month	num days	
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