Decision Structures and Boolean Logic

Topics

- The `if` Statement
- The `if-else` Statement
- Comparing Strings
- Nested Decision Structures and the `if-elif-else` Statement
- Logical Operators
- Boolean Variables
- Turtle Graphics: Determining the State of the Turtle

The `if` Statement (1 of 4)

- Control structure: logical design that controls order in which set of statements execute
- Sequence structure: set of statements that execute in the order they appear
- Decision structure: specific action(s) performed only if a condition exists
  - Also known as selection structure

The `if` Statement (2 of 4)

- In flowchart, diamond represents true/false condition that must be tested
- Actions can be *conditionally executed*
  - Performed only when a condition is true
- Single alternative decision structure: provides only one alternative path of execution
  - If condition is not true, exit the structure

The `if` Statement (3 of 4)

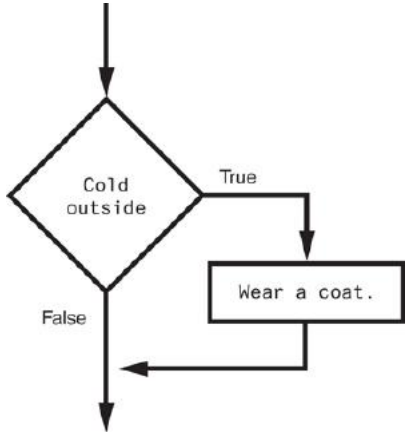


Figure 3-1 A simple decision structure

The `if` Statement (4 of 4)

- Python syntax:

```
if condition:
        Statement
        Statement
```

- First line known as the `if` clause
  - Includes the keyword `if` followed by condition
    - The condition can be true or false
    - When the `if` statement executes, the condition is tested, and if it is true the block statements are executed. otherwise, block statements are skipped

Boolean Expressions and Relational Operators (1 of 5)

- Boolean expression: expression tested by if statement to determine if it is true or false
  - Example: a > b
    - `true` if a is greater than b; `false` otherwise
- Relational operator: determines whether a specific relationship exists between two values
  - Example: greater than (>)

Boolean Expressions and Relational Operators (2 of 5)

- >= and <= operators test more than one relationship
  - It is enough for one of the relationships to exist for the expression to be true
- == operator determines whether the two operands are equal to one another
  - Do not confuse with assignment operator (=)
- != operator determines whether the two operands are not equal

Boolean Expressions and Relational Operators (3 of 5)

Table 3-2 Boolean expressions using relational operators

Expression	Meaning
`x > y`	Is x greater than y?
`x < y`	Is x less than y?
`x >= y`	Is x greater than or equal to y?
`x <= y`	Is x less than or equal to y?
`x == y`	Is x equal to y?
`x != y`	Is x not equal to y?

Boolean Expressions and Relational Operators (4 of 5)

- Using a Boolean expression with the > relational operator

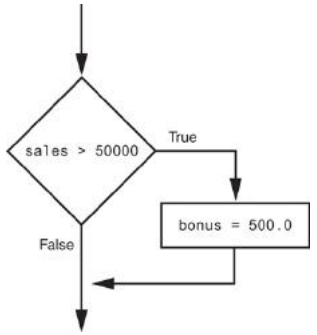


Figure 3-3 Example decision structure

Boolean Expressions and Relational Operators (5 of 5)

- Any relational operator can be used in a decision block
  - Example: `if balance == 0`
  - Example: `if payment != balance`
- It is possible to have a block inside another block
  - Example: `if` statement inside a function
  - Statements in inner block must be indented with respect to the outer block

The `if-else` Statement (1 of 3)

- Dual alternative decision structure: two possible paths of execution
  - One is taken if the condition is true, and the other if the condition is false
  - Syntax:

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

  - `if` clause and `else` clause must be aligned
  - Statements must be consistently indented

The `if-else` Statement (2 of 3)

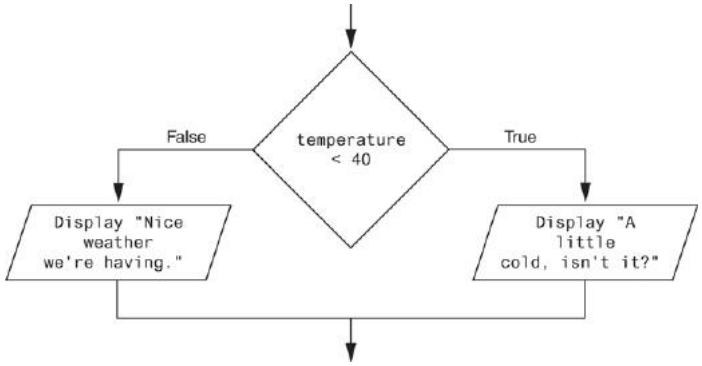


Figure 3-5 A dual alternative decision structure

The `if-else` Statement (3 of 3)

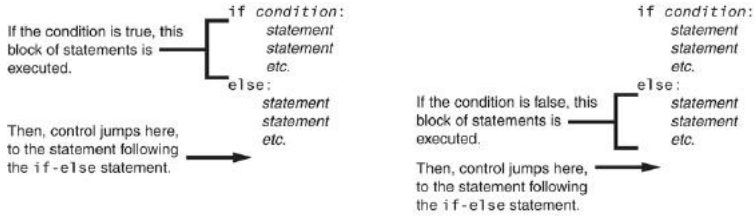


Figure 3-6 Conditional execution in an `if-else` statement

Comparing Strings (1 of 2)

- Strings can be compared using the == and != operators
- String comparisons are case sensitive
- Strings can be compared using >, <, >=, and <=
  - Compared character by character based on the ASCII values for each character
  - If shorter word is substring of longer word, longer word is greater than shorter word

Comparing Strings (2 of 2)

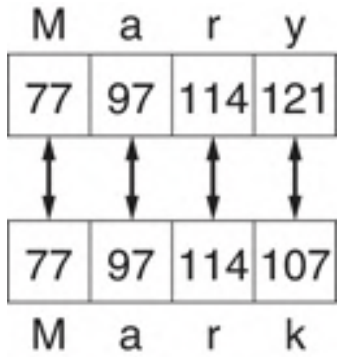


Figure 3-9 Comparing each character in a string

Nested Decision Structures and the `if-elif-else` Statement (1 of 3)

- A decision structure can be nested inside another decision structure
  - Commonly needed in programs
  - Example:
    - Determine if someone qualifies for a loan, they must meet two conditions:
      - Must earn at least $30,000/year
      - Must have been employed for at least two years
    - Check first condition, and if it is true, check second condition

Nested Decision Structures and the `if-elif-else` Statement (2 of 3)

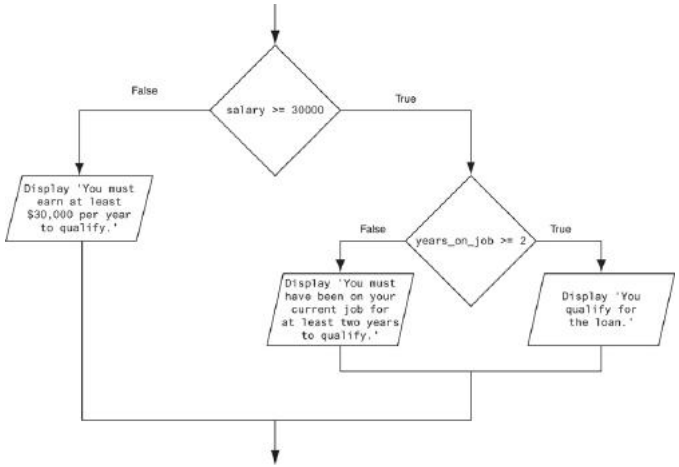


Figure 3-12 A nested decision structure

Nested Decision Structures and the `if-elif-else` Statement (3 of 3)

- Important to use proper indentation in a nested decision structure
  - Important for Python interpreter
  - Makes code more readable for programmer
  - Rules for writing nested if statements:
    - `else` clause should align with matching `if` clause
    - Statements in each block must be consistently indented

The `if-elif-else` Statement (1 of 3)

- `if-elif-else` statement: special version of a decision structure
  - Makes logic of nested decision structures simpler to write
    - Can include multiple `elif` statements
  - Syntax:

```
if condition_1:
    statement(s)
elif condition_2:
    statement(s)
elif condition_3:
    statement(s)
else
    statement(s)
```

Insert as many `elif` clauses as necessary.

The `if-elif-else` Statement (2 of 3)

- Alignment used with `if-elif-else` statement:
  - `if`, `elif`, and `else` clauses are all aligned
  - Conditionally executed blocks are consistently indented
- `if-elif-else` statement is never required, but logic easier to follow
  - Can be accomplished by nested `if-else`
    - Code can become complex, and indentation can cause problematic long lines

The `if-elif-else` Statement (3 of 3)

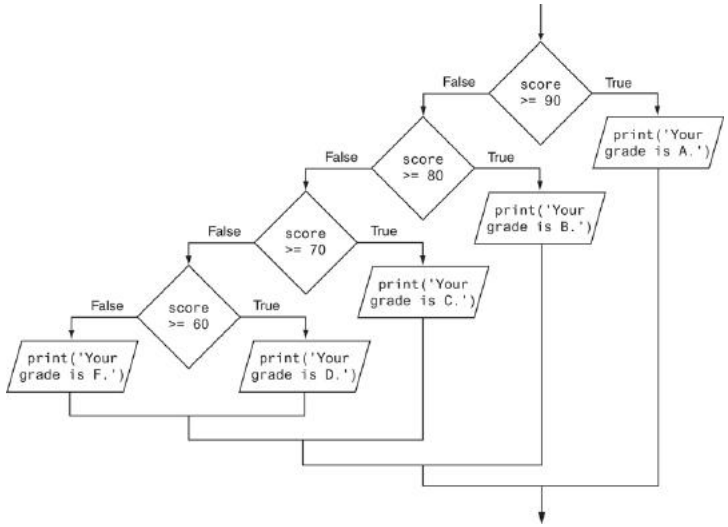


Figure 3-15 Nested decision structure to determine a grade

Logical Operators

- Logical operators: operators that can be used to create complex Boolean expressions
  - `and` operator and `or` operator: binary operators, connect two Boolean expressions into a compound Boolean expression
  - `not` operator: unary operator, reverses the truth of its Boolean operand

The and Operator

- Takes two Boolean expressions as operands
  - Creates compound Boolean expression that is true only when both sub expressions are true
  - Can be used to simplify nested decision structures
- Truth table for the `and` operator

Expression	Value of the Expression
false and false	false
false and true	false
true and false	false
true and true	true

The `or` Operator

- Takes two Boolean expressions as operands
  - Creates compound Boolean expression that is true when either of the sub expressions is true
  - Can be used to simplify nested decision structures
- Truth table for the `or` operator

Expression	Value of the Expression
false or false	false
false or true	true
true or false	true
true or true	true

Short-Circuit Evaluation

- Short circuit evaluation: deciding the value of a compound Boolean expression after evaluating only one sub expression
  - Performed by the `or` and `and` operators
    - For `or` operator: If left operand is true, compound expression is true. Otherwise, evaluate right operand
    - For `and` operator: If left operand is false, compound expression is false. Otherwise, evaluate right operand

The `not` Operator

- Takes one Boolean expressions as operand and reverses its logical value
  - Sometimes it may be necessary to place parentheses around an expression to clarify to what you are applying the not operator
- Truth table for the `not` operator

Expression	Value of the Expression
true	false
false	true

Checking Numeric Ranges with Logical Operators

- To determine whether a numeric value is within a specific range of values, use `and`
  - Example: `x >= 10 and x <= 20`
- To determine whether a numeric value is outside of a specific range of values, use `or`
  - Example: `x < 10 or x > 20`

Boolean Variables

- Boolean variable: references one of two values, `True` or `False`
  - Represented by `bool` data type
- Commonly used as flags
  - Flag: variable that signals when some condition exists in a program
    - Flag set to `False` → condition does not exist
    - Flag set to `True` → condition exists

Turtle Graphics: Determining the State of the Turtle (1 of 9)

- The `turtle.xcor()` and `turtle.ycor()` functions return the turtle's *X* and *Y* coordinates
- Examples of calling these functions in an `if` statement:

```
if turtle.ycor() < 0:
    turtle.goto(0, 0)


if turtle.xcor() > 100 and turtle.xcor() < 200:
    turtle.goto(0, 0)
```


Turtle Graphics: Determining the State of the Turtle (2 of 9)

- The `turtle.heading()` function returns the turtle's heading. (By default, the heading is returned in degrees.)
- Example of calling the function in an `if` statement:

```
if turtle.heading() >= 90 and turtle.heading() <= 270:
    turtle.setheading(180)
```


Turtle Graphics: Determining the State of the Turtle (3 of 9)

- The `turtle.isdown()` function returns `True` if the pen is down, or `False` otherwise.
- Example of calling the function in an `if` statement:

```
if turtle.isdown():
    turtle.penup()


if not(turtle.isdown()):
    turtle.pendown()
```

Turtle Graphics: Determining the State of the Turtle (4 of 9)

- The `turtle.isvisible()` function returns `True` if the turtle is visible, or `False` otherwise.
- Example of calling the function in an `if` statement:

```
if turtle.isvisible():
    turtle.hideturtle()
```


Turtle Graphics: Determining the State of the Turtle (5 of 9)

- When you call `turtle.pencolor()` without passing an argument, the function returns the pen's current color as a string. Example of calling the function in an `if` statement:

```
if turtle.pencolor() == 'red':
    turtle.pencolor('blue')
```

- When you call `turtle.fillcolor()` without passing an argument, the function returns the current fill color as a string. Example of calling the function in an `if` statement:

```
if turtle.fillcolor() ==
'blue':
    turtle.fillcolor('white')
```


Turtle Graphics: Determining the State of the Turtle (6 of 9)

- When you call `turtle.bgcolor()` without passing an argument, the function returns the current background color as a string. Example of calling the function in an `if` statement:

```
if turtle.bgcolor() == 'white':
    turtle.bgcolor('gray')
```


Turtle Graphics: Determining the State of the Turtle (7 of 9)

- When you call `turtle.pensize()` without passing an argument, the function returns the pen's current size as a string. Example of calling the function in an `if` statement:

```
if turtle.pensize() < 3:
    turtle.pensize(3)
```

Turtle Graphics: Determining the State of the Turtle (8 of 9)

- When you call `turtle.speed()` without passing an argument, the function returns the current animation speed. Example of calling the function in an `if` statement:

```
if turtle.speed() > 0:
    turtle.speed(0)
```


Turtle Graphics: Determining the State of the Turtle (9 of 9)

- See *In the Spotlight: The Hit the Target Game* in your textbook for numerous examples of determining the state of the turtle.

Summary

- This chapter covered:
  - Decision structures, including:
    - Single alternative decision structures
    - Dual alternative decision structures
    - Nested decision structures
  - Relational operators and logical operators as used in creating Boolean expressions
  - String comparison as used in creating Boolean expressions
  - Boolean variables
  - Determining the state of the turtle in Turtle Graphics

Repetition Structures

Topics

- Introduction to Repetition Structures
- The `while` Loop: a Condition-Controlled Loop
- The `for` Loop: a Count-Controlled Loop
- Calculating a Running Total
- Sentinels
- Input Validation Loops
- Nested Loops
- Turtle Graphics: Using Loops to Draw Designs

Introduction to Repetition Structures

- Often have to write code that performs the same task multiple times
  - Disadvantages to duplicating code
    - Makes program large
    - Time consuming
    - May need to be corrected in many places
- Repetition structure: makes computer repeat included code as necessary
  - Includes condition-controlled loops and count-controlled loops

The `while` Loop: a Condition-Controlled Loop (1 of 4)

- `while` loop: while condition is true, do something
  - Two parts:
    - Condition tested for true or false value
    - Statements repeated as long as condition is true
  - In flow chart, line goes back to previous part
  - General format:

```
while condition:
    statements
```


The `while` Loop: a Condition-Controlled Loop (2 of 4)

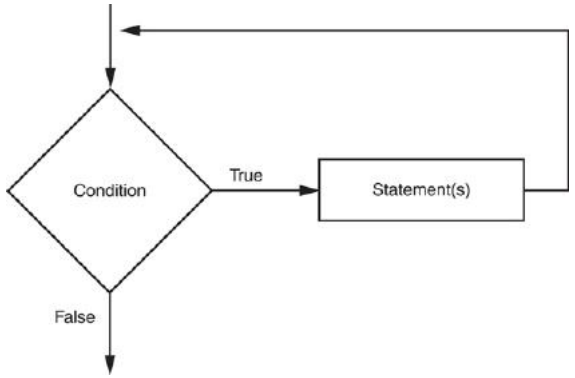


Figure 4-1 The logic of a `while` loop

The `while` Loop: a Condition-Controlled Loop (3 of 4)

- In order for a loop to stop executing, something has to happen inside the loop to make the condition false
- Iteration: one execution of the body of a loop
- `while` loop is known as a *pretest* loop
  - Tests condition before performing an iteration
    - Will never execute if condition is false to start with
    - Requires performing some steps prior to the loop

The `while` Loop: a Condition-Controlled Loop (4 of 4)

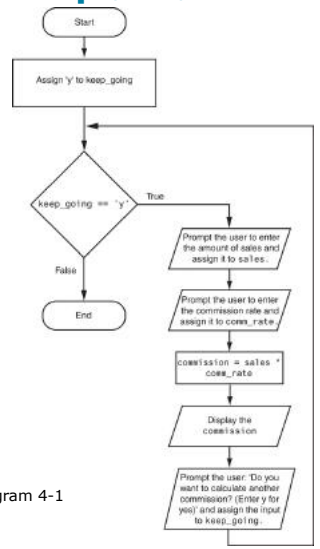


Figure 4-3 Flowchart for Program 4-1

Infinite Loops

- Loops must contain within themselves a way to terminate
  - Something inside a `while` loop must eventually make the condition false
- Infinite loop: loop that does not have a way of stopping
  - Repeats until program is interrupted
  - Occurs when programmer forgets to include stopping code in the loop

The `for` Loop: a Count-Controlled Loop (1 of 2)

- Count-Controlled loop: iterates a specific number of times
  - Use a `for` statement to write count-controlled loop
    - Designed to work with sequence of data items
      - Iterates once for each item in the sequence
    - General format:

```
for variable in [val1, val2, etc]:
    statements
```

    - Target variable: the variable which is the target of the assignment at the beginning of each iteration

The `for` Loop: a Count-Controlled Loop
(2 of 2)

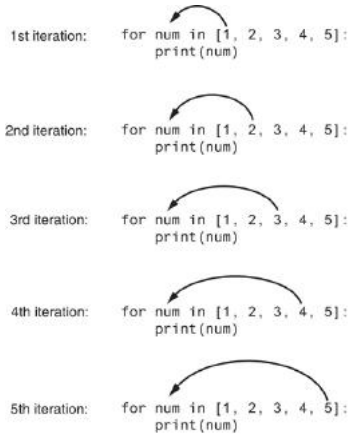


Figure 4-4 The `for` loop

Using the `range` Function with the `for` Loop

- The `range` function simplifies the process of writing a `for` loop
  - `range` returns an iterable object
    - Iterable: contains a sequence of values that can be iterated over
- `range` characteristics:
  - One argument: used as ending limit
  - Two arguments: starting value and ending limit
  - Three arguments: third argument is step value

Using the Target Variable Inside the Loop

- Purpose of target variable is to reference each item in a sequence as the loop iterates
- Target variable can be used in calculations or tasks in the body of the loop
  - Example: calculate square root of each number in a range

Letting the User Control the Loop Iterations

- Sometimes the programmer does not know exactly how many times the loop will execute
- Can receive range inputs from the user, place them in variables, and call the `range` function in the for clause using these variables
  - Be sure to consider the end cases: `range` does not include the ending limit

Generating an Iterable Sequence that Ranges from Highest to Lowest

- The `range` function can be used to generate a sequence with numbers in descending order
  - Make sure starting number is larger than end limit, and step value is negative
  - Example: `range(10, 0, -1)`

Calculating a Running Total (1 of 2)

- Programs often need to calculate a total of a series of numbers
  - Typically include two elements:
    - A loop that reads each number in series
    - An *accumulator* variable
  - Known as program that keeps a running total: accumulates total and reads in series
  - At end of loop, accumulator will reference the total

Calculating a Running Total (2 of 2)

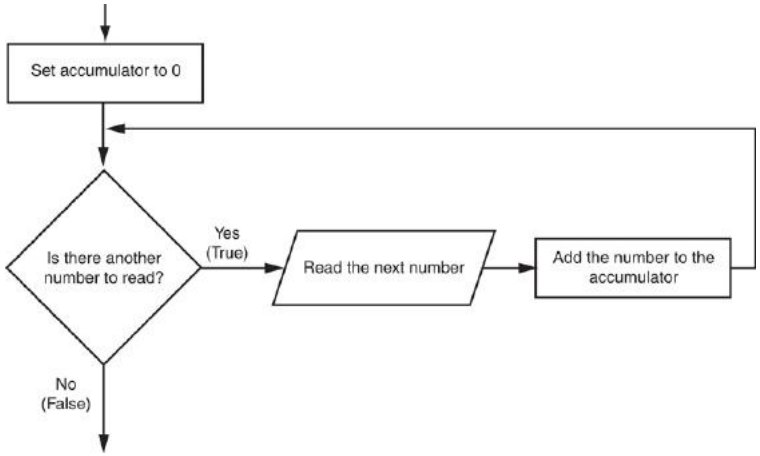


Figure 4-6 Logic for calculating a running total

The Augmented Assignment Operators (1 of 2)

- In many assignment statements, the variable on the left side of the = operator also appears on the right side of the = operator
- Augmented assignment operators: special set of operators designed for this type of job
  - Shorthand operators

The Augmented Assignment Operators
(2 of 2)

Table 4-2 Augmented assignment operators

Operator	Example Usage	Equivalent To
`+=`	`x += 5`	`x = x + 5`
`-=`	`y -= 2`	`y = y - 2`
`*=`	`z *= 10`	`z = z * 10`
`/=`	`a /= b`	`a = a / b`
`%=`	`c %= 3`	`c = c % 3`
`//=`	`x //= 3`	`x = x // 3`
`**=`	`y **= 2`	`y = y**2`

Sentinels

- Sentinel: special value that marks the end of a sequence of items
  - When program reaches a sentinel, it knows that the end of the sequence of items was reached, and the loop terminates
  - Must be distinctive enough so as not to be mistaken for a regular value in the sequence
  - Example: when reading an input file, empty line can be used as a sentinel

Input Validation Loops (1 of 3)

- Computer cannot tell the difference between good data and bad data
  - If user provides bad input, program will produce bad output
  - GIGO: garbage in, garbage out
  - It is important to design program such that bad input is never accepted

Input Validation Loops (2 of 3)

- Input validation: inspecting input before it is processed by the program
  - If input is invalid, prompt user to enter correct data
  - Commonly accomplished using a `while` loop which repeats as long as the input is bad
    - If input is bad, display error message and receive another set of data
    - If input is good, continue to process the input

Input Validation Loops (3 of 3)

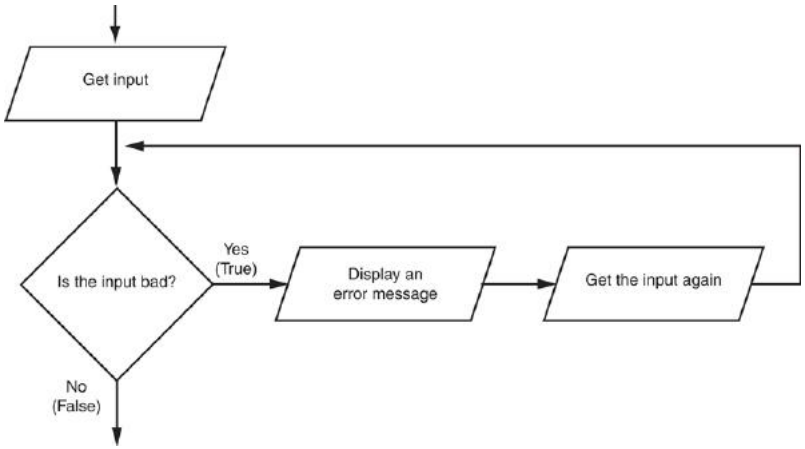


Figure 4-7 Logic containing an input validation loop

Nested Loops (1 of 3)

- Nested loop: loop that is contained inside another loop
  - Example: analog clock works like a nested loop
    - Hours hand moves once for every twelve movements of the minutes hand: for each iteration of the "hours," do twelve iterations of "minutes"
    - Seconds hand moves 60 times for each movement of the minutes hand: for each iteration of "minutes," do 60 iterations of "seconds"

Nested Loops (2 of 3)

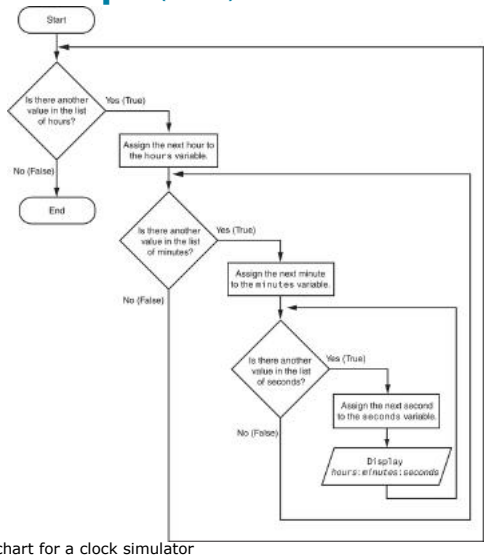


Figure 4-8 Flowchart for a clock simulator

Nested Loops (3 of 3)

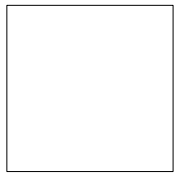
- Key points about nested loops:
  - Inner loop goes through all of its iterations for each iteration of outer loop
  - Inner loops complete their iterations faster than outer loops
  - Total number of iterations in nested loop:

number of iterations of inner loop X *number of iterations of outer loop*

Turtle Graphics: Using Loops to Draw Designs (1 of 4)

- You can use loops with the turtle to draw both simple shapes and elaborate designs. For example, the following for loop iterates four times to draw a square that is 100 pixels wide:

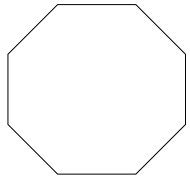
```
for x in range(4):
    turtle.forward(100)
    turtle.right(90)
```



Turtle Graphics: Using Loops to Draw Designs (2 of 4)

- This `for` loop iterates eight times to draw the octagon:

```
for x in range(8):
    turtle.forward(100)
    turtle.right(45)
```

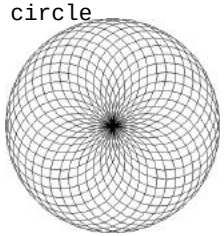


Turtle Graphics: Using Loops to Draw Designs (3 of 4)

- You can create interesting designs by repeatedly drawing a simple shape, with the turtle tilted at a slightly different angle each time it draws the shape.

```
NUM_CIRCLES = 36    # Number of circles to
draw
RADIUS = 100        # Radius of each circle
ANGLE = 10          # Angle to turn

for x in range(NUM_CIRCLES):
    turtle.circle(RADIUS)
    turtle.left(ANGLE)
```



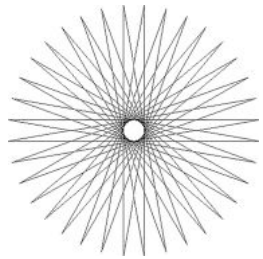
Turtle Graphics: Using Loops to Draw Designs (4 of 4)

- This code draws a sequence of 36 straight lines to make a "starburst" design.

```
START_X = -200      # Starting X coordinate
START_Y = 0         # Starting Y coordinate
NUM_LINES = 36      # Number of lines to draw
LINE_LENGTH = 400   # Length of each line
ANGLE = 170         # Angle to turn

turtle.hideturtle()
turtle.penup()
turtle.goto(START_X, START_Y)
turtle.pendown()

for x in range(NUM_LINES):
    turtle.forward(LINE_LENGTH)
    turtle.left(ANGLE)
```



Summary

- This chapter covered:
  - Repetition structures, including:
    - Condition-controlled loops
    - Count-controlled loops
    - Nested loops
  - Infinite loops and how they can be avoided
  - `range` function as used in `for` loops
  - Calculating a running total and augmented assignment operators
  - Use of sentinels to terminate loops
  - Using loops to draw turtle graphic designs