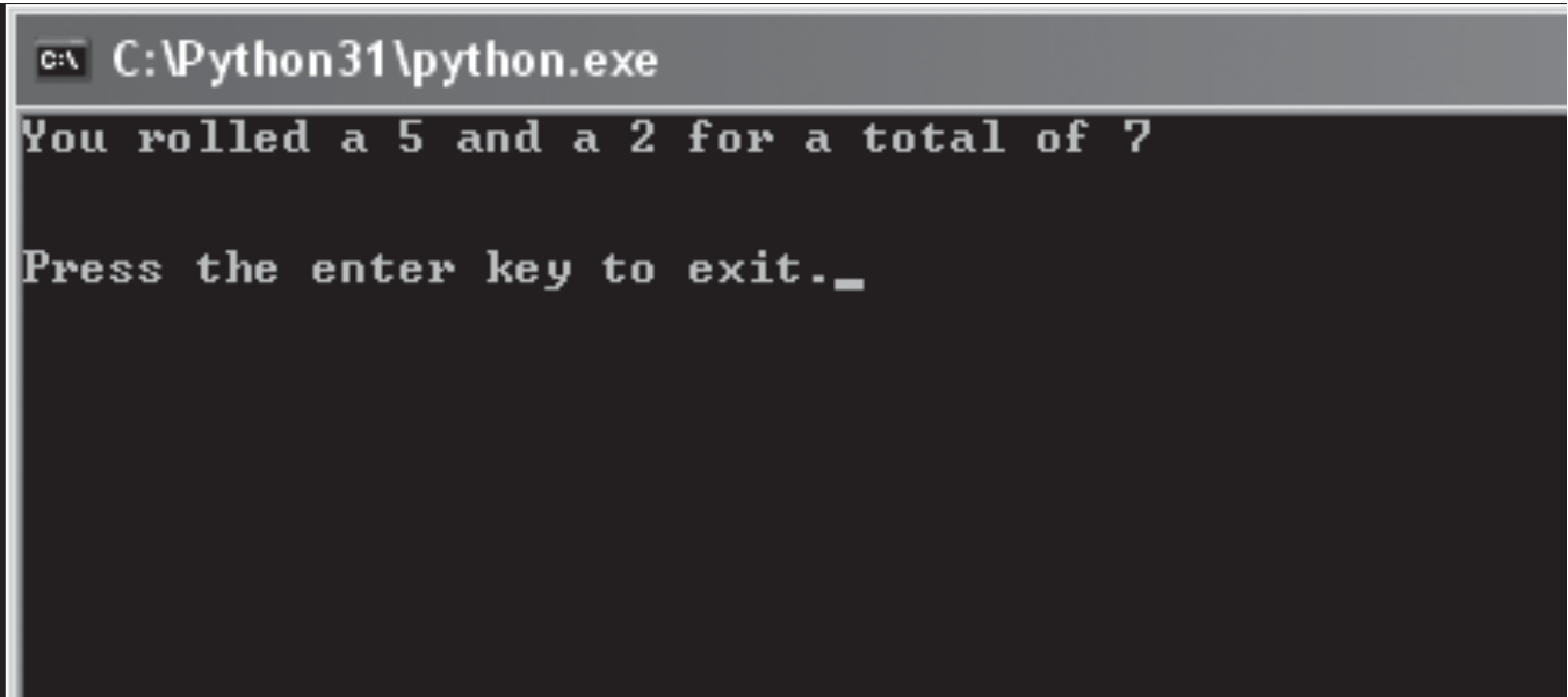
Chapter 3

Branching, while Loops, And Program Planning: The Guess My Number Game

Introducing the Craps Roller Program

- Craps Roller simulates the roll of 2 6-sided dices, and displays the value of each and their total.

- To determine the dice values, the program uses functions that generate random numbers.



```
You rolled a 5 and a 2 for a total of 7


Press the enter key to exit._
```

craps_roller.py

```
# Craps Roller
# Demonstrates random number generation

import random

# generate random numbers 1 - 6

die1 = random.randint(1, 6)
die2 = random.randrange(6) + 1

total = die1 + die2

print("You rolled a", die1, "and a", die2,
          "for a total of", total)

input("\n\nPress the enter key to exit.")
```

Importing the random Module

- The **import** statement allows you to import, or load, modules, eg,

import random

- *Modules* are files that contain code meant to be used in other programs. These modules usually group together a collection of programming related to one area.

- The **random** module contains functions related to generating random numbers and producing random results.

Using the randint() Function

- The random module contains a function, **randint()** , which produces a random **integer**.

- The program accesses **randint()** through the following function call:

  random.randint(1, 6)

- You can call a function from an imported module by giving the module name, followed by a **period**, then the function call itself. This method of access is called *dot notation*.

- random.**randint()** means the function **randint()** that belongs to the module random. Dot notation can be used to access different elements of imported modules.

- **randint()** requires 2 integer argument values and returns a random integer between those 2 values, which may include either of the argument values.

Using the randrange() Function

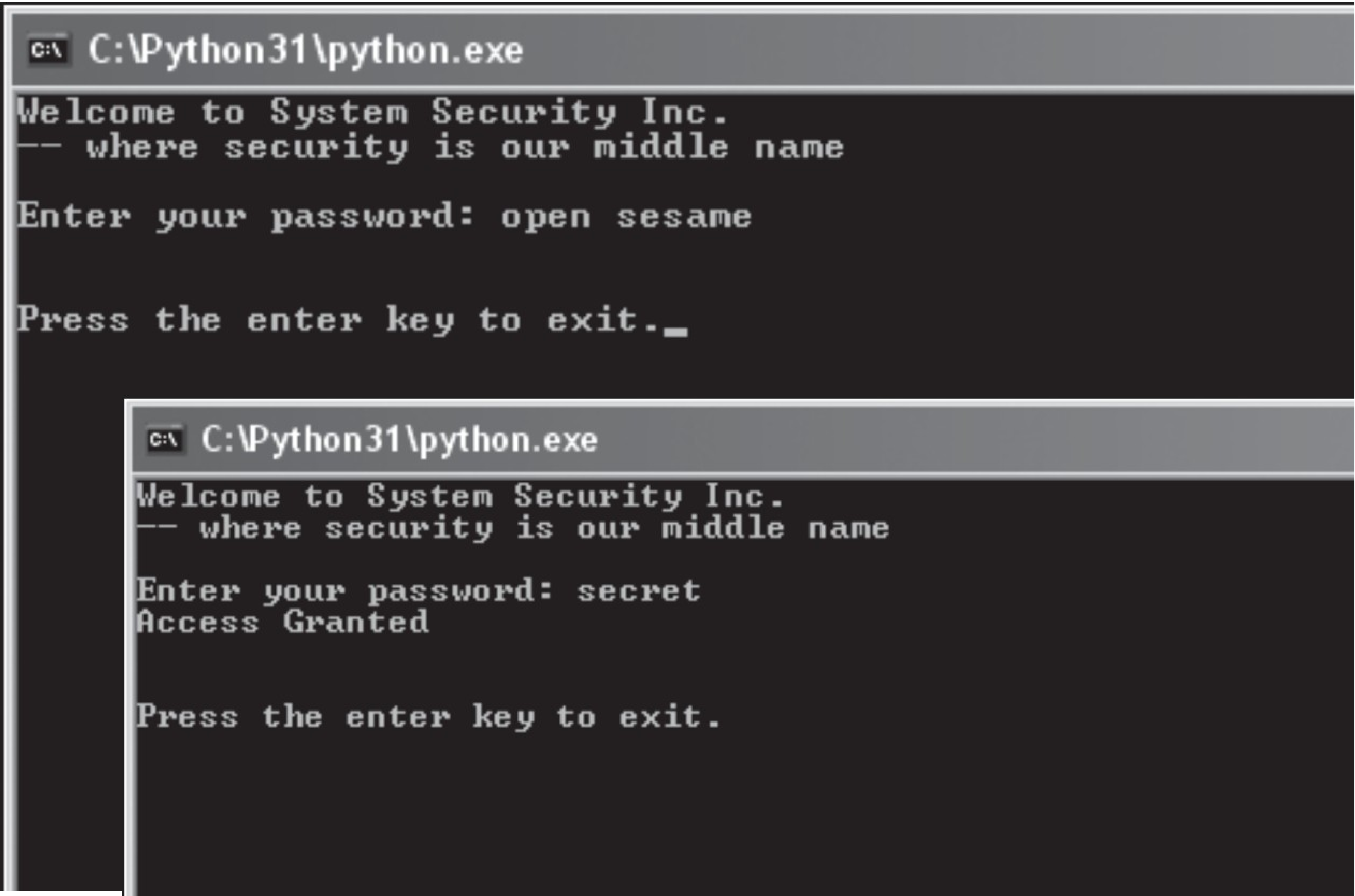
- The random module also contains **randrange()**, which produces a random integer.

- random.**randrange(N)** gives an integer in {0, …, N–1}

- To produces either a 1, 2, 3, 4, 5, or 6

```
die2 = random.randrange(6) + 1
```

Introducing the Password Program



C:\Python31\python.exe

```
Welcome to System Security Inc.
-- where security is our middle name

Enter your password: open sesame


Press the enter key to exit._
```

C:\Python31\python.exe

```
Welcome to System Security Inc.
-- where security is our middle name

Enter your password: secret
Access Granted


Press the enter key to exit.
```

password.py

```
# Password
# Demonstrates the if statement

print("Welcome to System Security Inc.")
print("-- where security is our middle name\n")

password = input("Enter your password: ")

if password == "secret":
    print("Access Granted")

input("\n\nPress the enter key to exit.")
```

Examining the if Statement

- The key to program Password is the **if** statement:

```
if password == "secret":
    print("Access Granted")
```

- If password is equal to "secret" , then "Access Granted" is printed and the program continues to the next statement.

- If it isn’t equal to "secret" , the program does not print the message and continues directly to the next statement.

Creating Conditions

- All if statements have a *condition*. A condition is just an expression that is either **true** or **false**.

- A condition can always be evaluated to be either **True** (representing true) and **False** (representing false).

- The condition in the if statement is **password=="secret"** . It means that password is equal to "secret" .

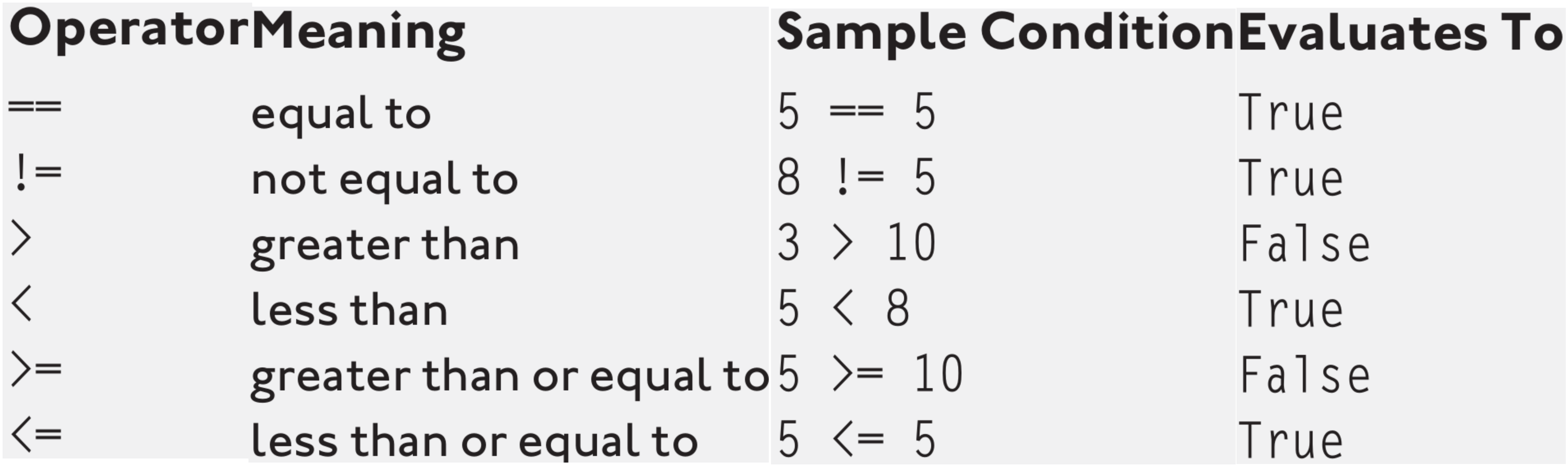
- This condition evaluates to either True or False , depending on the value of password .

Understanding Comparison Operators

- Conditions are often created by comparing values. You can compare values using *comparison operators*.

- The equal-to comparison operator is written as ==.

- password = "secret" is an assignment statement. It assigns a value. password == "secret" is a condition. It evaluates to either True or False .



Operator	Meaning	Sample Condition	Evaluates To
==	equal to	5 == 5	True
!=	not equal to	8 != 5	True
>	greater than	3 > 10	False
<	less than	5 < 8	True
>=	greater than or equal to	5 >= 10	False
<=	less than or equal to	5 <= 5	True

- you can compare integers to integers, floating-point numbers to floating-point numbers, as well as integers to floating-point numbers.

- You can even compare strings—the results are based on alphabetical order.

- "apple"<"orange" is True because "apple" is alphabetically less than "orange".

- Objects of different types that don’t have an established definition for order can’t be compared using the **<**, **<=**, **>**, or **>=** operators.

- Python won’t let you use these operators to compare strings and integers, eg, 10 > "five".

Using Indentation to Create Blocks

- By indenting the line, it becomes a *block*. A block is one or more consecutive lines indented by the **same** amount.

- Blocks can be used as part of an if statement. They're the statement or group of statements that gets executed if the condition is True .

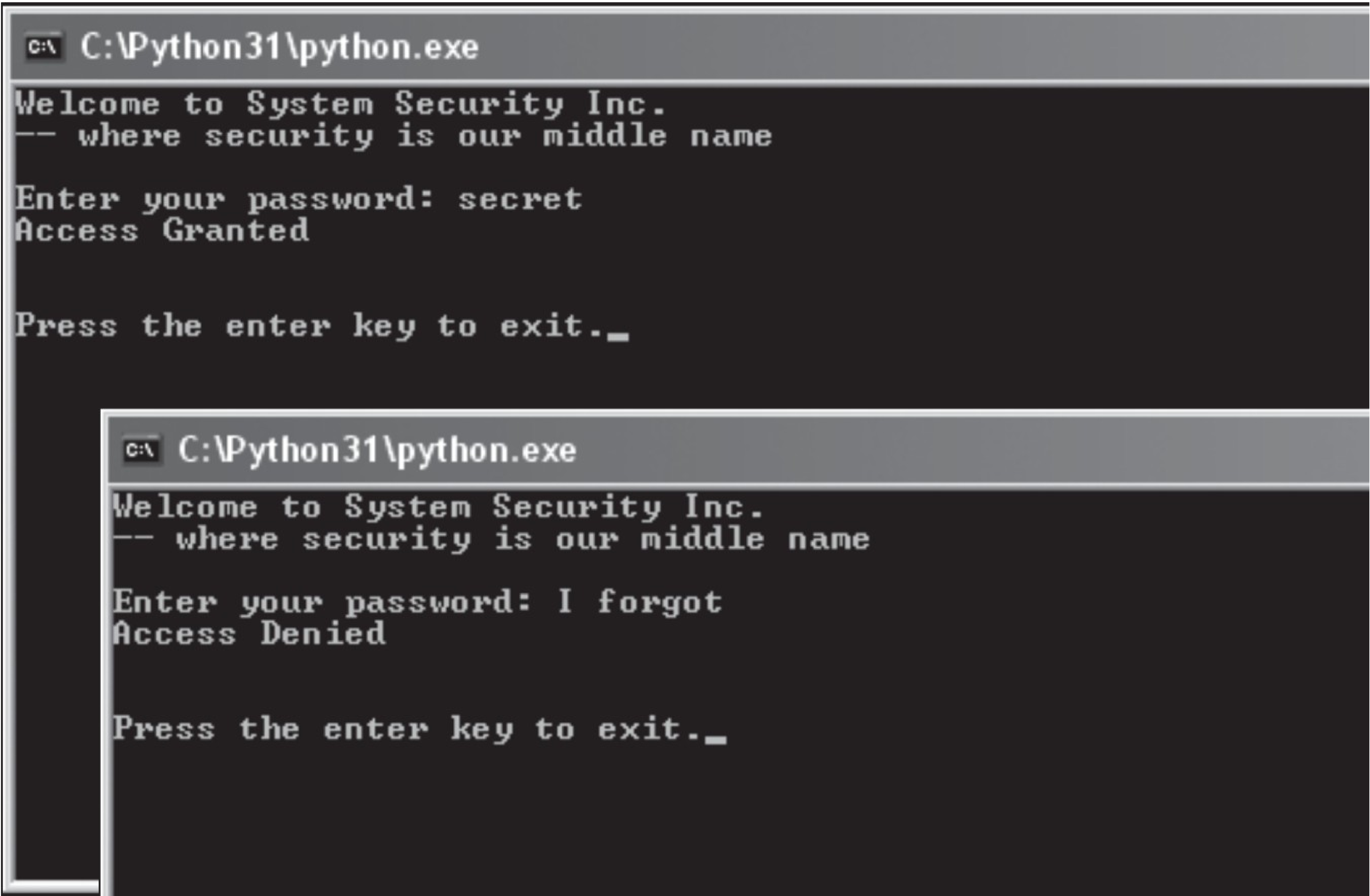
- you could add more statements in the block:

```
if password == "secret":
    print("Access Granted")
    print("Welcome! You must be very important.")
```

- There's debate about whether to use tabs or spaces. There are 2 guidelines:

  1. be consistent
  2. don't mix spaces and tabs.

The Granted or Denied Program


C:\Python31\python.exe

```
Welcome to System Security Inc.
-- where security is our middle name

Enter your password: secret
Access Granted


Press the enter key to exit._
```

C:\Python31\python.exe

```
Welcome to System Security Inc.
-- where security is our middle name

Enter your password: I forgot
Access Denied


Press the enter key to exit._
```

granted_or_denied.py

```
# Granted or Denied
# Demonstrates an else clause

print("Welcome to System Security Inc.")
print("-- where security is our middle name\n")

password = input("Enter your password: ")

if password == "secret":
    print("Access Granted")
else:
    print("Access Denied")

input("\n\nPress the enter key to exit.")
```

Examining the else Clause

- added an **else** clause to the if statement:

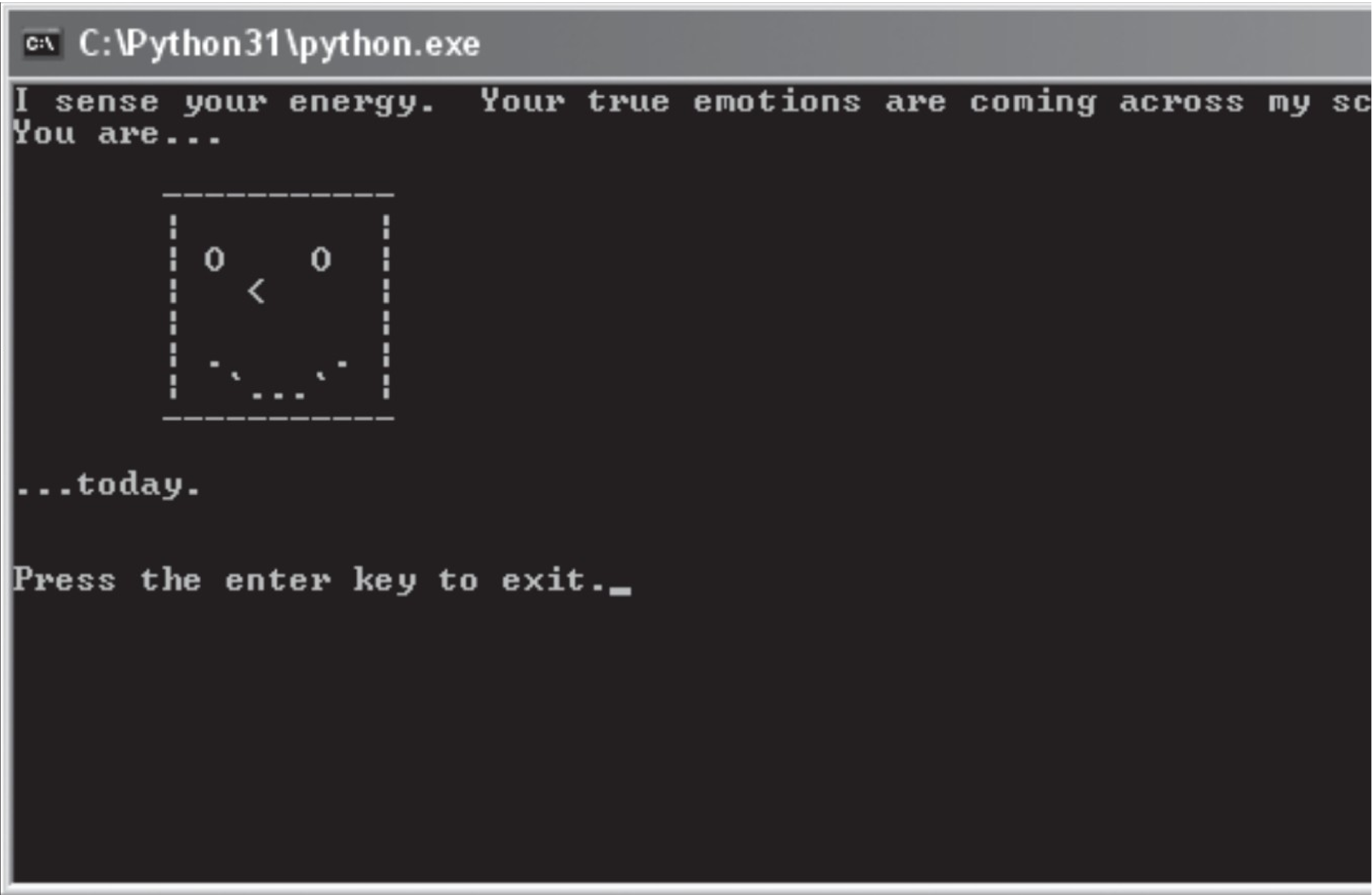
```
if password == "secret":
    print("Access Granted")
else:
    print("Access Denied")
```

- If the value of password is equal to "secret" , the program prints Access Granted , just like before. And the program prints Access Denied otherwise.

- In an if statement with an **else** clause, exactly one of the code blocks will execute.

- You can create an **else** clause immediately following the if block with **else**, followed by a colon, followed by a block of statements. The **else** clause must be in the same block as its corresponding if, ie, the **else** and if must be indented the same amount.

Introducing the Mood Computer Program



```
I sense your energy.  Your true emotions are coming across my sc
You are...

       -----------
       |         |
       | O     O |
       |    <    |
       |         |
       | .     . |
       |  `...`  |
       -----------

...today.


Press the enter key to exit._
```

mood_computer.py

```
# Mood Computer
# Demonstrates the elif clause

import random

print(“I sense your energy.  Your true emotions “,
       “are coming across my screen.”)
print("You are...")

mood = random.randint(1, 3)

if mood == 1:
    # happy
    print(
    """
```

```
      ------------
      |          |
      | O    O  |
      |    <      |
      |          |
      | .      .  |
      |  `...`   |
      ------------
                    """)
elif mood == 2:
    # neutral
    print(
    """
      ------------
      |          |
      | O    O  |
      |    <     |
      |          |
      | --------  |
      |          |
      ------------
```

```
                    """)
elif mood == 3:
    # sad
    print(
    """

       -------------
       |           |
       | O    O    |
       |    <     |
       |           |
       |   . ' .   |
       | '      '  |
       -------------
                    """)
else:
    print("Illegal mood value!”,
        “(You must be in a really bad mood).")

print("...today.")
input("\n\nPress the enter key to exit.")
```

Examining the elif Clause

- An if statement with **elif** (short for “else if”) clauses can contain a sequence of conditions for a program to evaluate.

- In the program, the lines containing the 3 conditions are

  - if mood == 0:
  - **elif** mood == 1:
  - **elif** mood == 2:

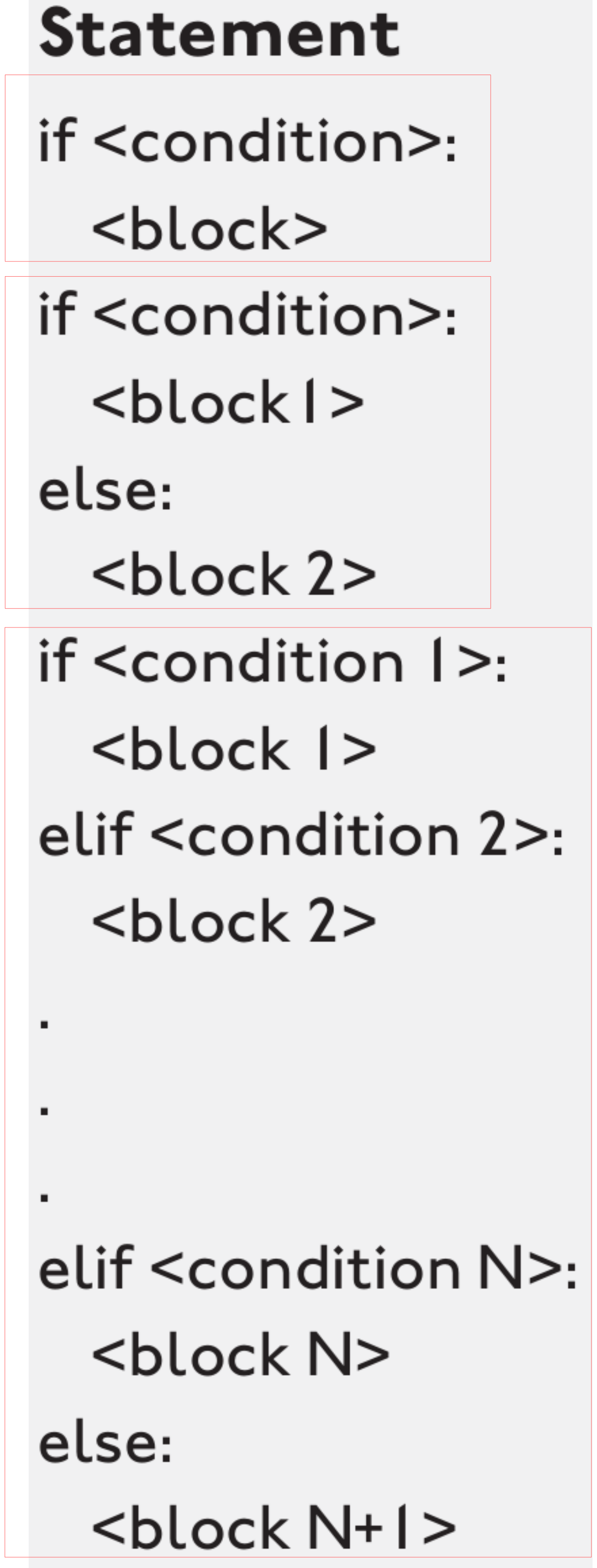
- You can have as many **elif** clauses as you like.

- An important feature of an if statement with **elif** clauses is that once a condition evaluates to True, the computer executes its corresponding block and exits the statement.

- It is possible to create statements where more than one condition can be true at the same time. In that case, only the block associated with the 1st true condition executes.

- If none of the preceding conditions for mood turn out to be True, then the final else clause's block runs.

- It's a good idea to use the final else clause. It works as a catchall for when none of the conditions within the statement are True.



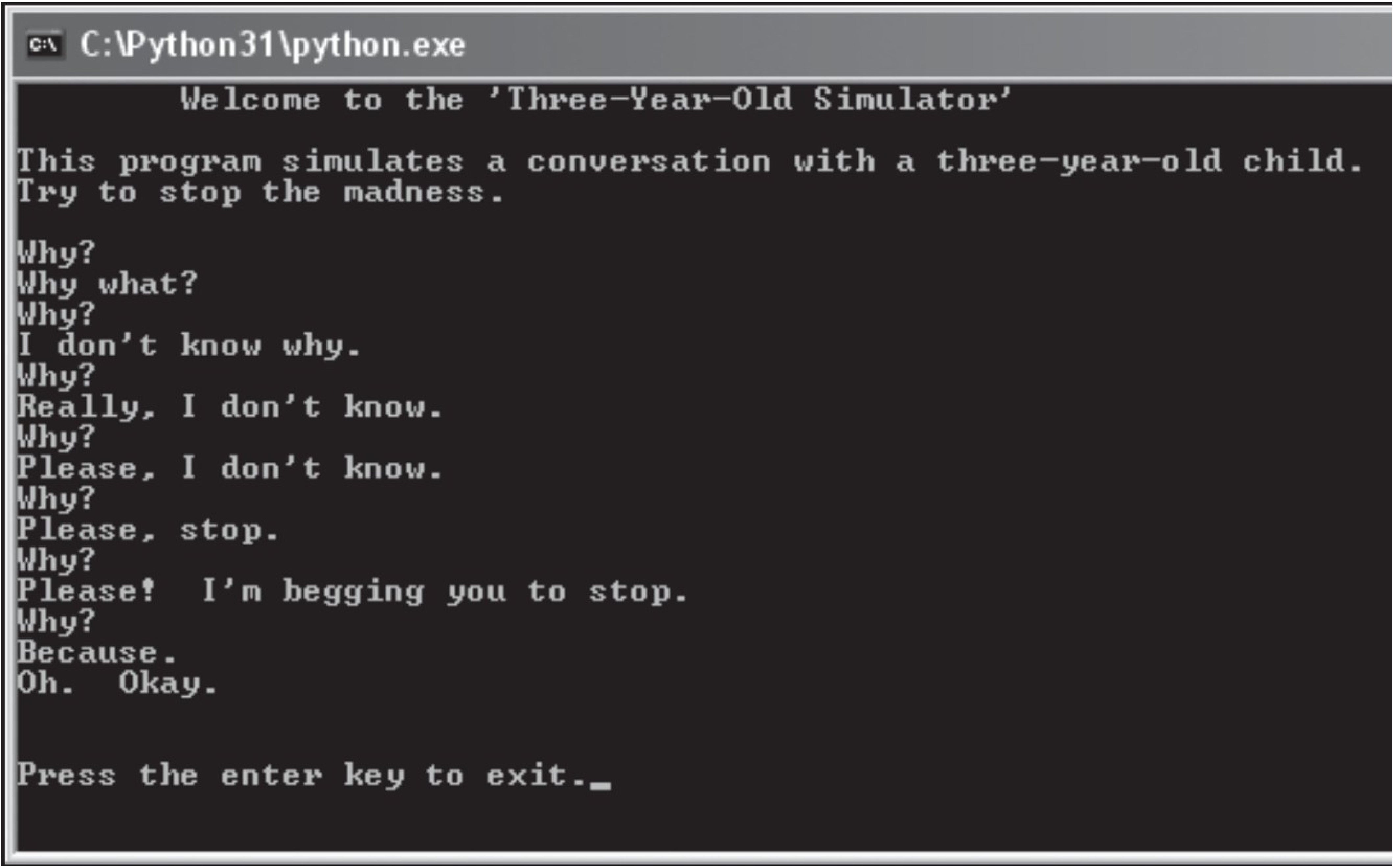
Statement

```
if <condition>:
    <block>
```

```
if <condition>:
    <block 1>
else:
    <block 2>
```

```
if <condition 1>:
    <block 1>
elif <condition 2>:
    <block 2>
.
.
.
elif <condition N>:
    <block N>
else:
    <block N+1>
```

The 3-Year-Old Simulator Program



```
        Welcome to the 'Three-Year-Old Simulator'

This program simulates a conversation with a three-year-old child.
Try to stop the madness.

Why?
Why what?
Why?
I don't know why.
Why?
Really, I don't know.
Why?
Please, I don't know.
Why?
Please, stop.
Why?
Please!  I'm begging you to stop.
Why?
Because.
Oh.  Okay.


Press the enter key to exit._
```

three_year-old.py

```
# Three Year-Old Simulator
# Demonstrates the while loop

print("\tWelcome to the 'Three-Year-Old Simulator'\n")
print("This program simulates a conversation with”,
        “a three-year-old child.")
print("Try to stop the madness.\n")

response = ""
while response != "Because.":
    response = input("Why?\n")

print("Oh.  Okay.")

input("\n\nPress the enter key to exit.")
```

Examining the while Loop

- The loop from the program is just 2 lines:

```
while response != "Because.":
      response = input("Why?\n")
```

- In the **while** statement, similar to the if statement, if the condition is true, the block (the *loop body*) is executed.

- In the **while** statement, the computer tests the condition and executes the block over and over, until the condition is false. That's why it's called a loop.

- The loop body is just response = input("Why?\n") , which will continue to execute until the user enters Because.

Initializing the Sentry Variable

- Often, while loops are controlled by a *sentry variable*, a variable used in the condition and compared to some other value or values.

- In the previous program, the sentry variable is response .

- It’s important to initialize your sentry variable. Most of the time, sentry variables are initialized right before the loop itself.

- The code initializes the sentry variable with response = ""

- If the sentry variable doesn’t have a value when the condition is evaluated, your program will generate an error.

- It’s usually a good idea to initialize your sentry variables to some type of empty value.

Checking the Sentry Variable

- Make sure that it’s possible for the while condition to evaluate to True at some point; otherwise, the block will never run.

- If you add one extra line to the program as

```
response = "Because."
while response != "Because.":
    response = input("Why?\n")
```

- Since response is equal to "Because." right before the loop, the block will never run.

Updating the Sentry Variable

- Once you’ve established your condition, initialized your sentry variable, and are sure that under some conditions the loop block will execute, then make sure the loop will end at some point.

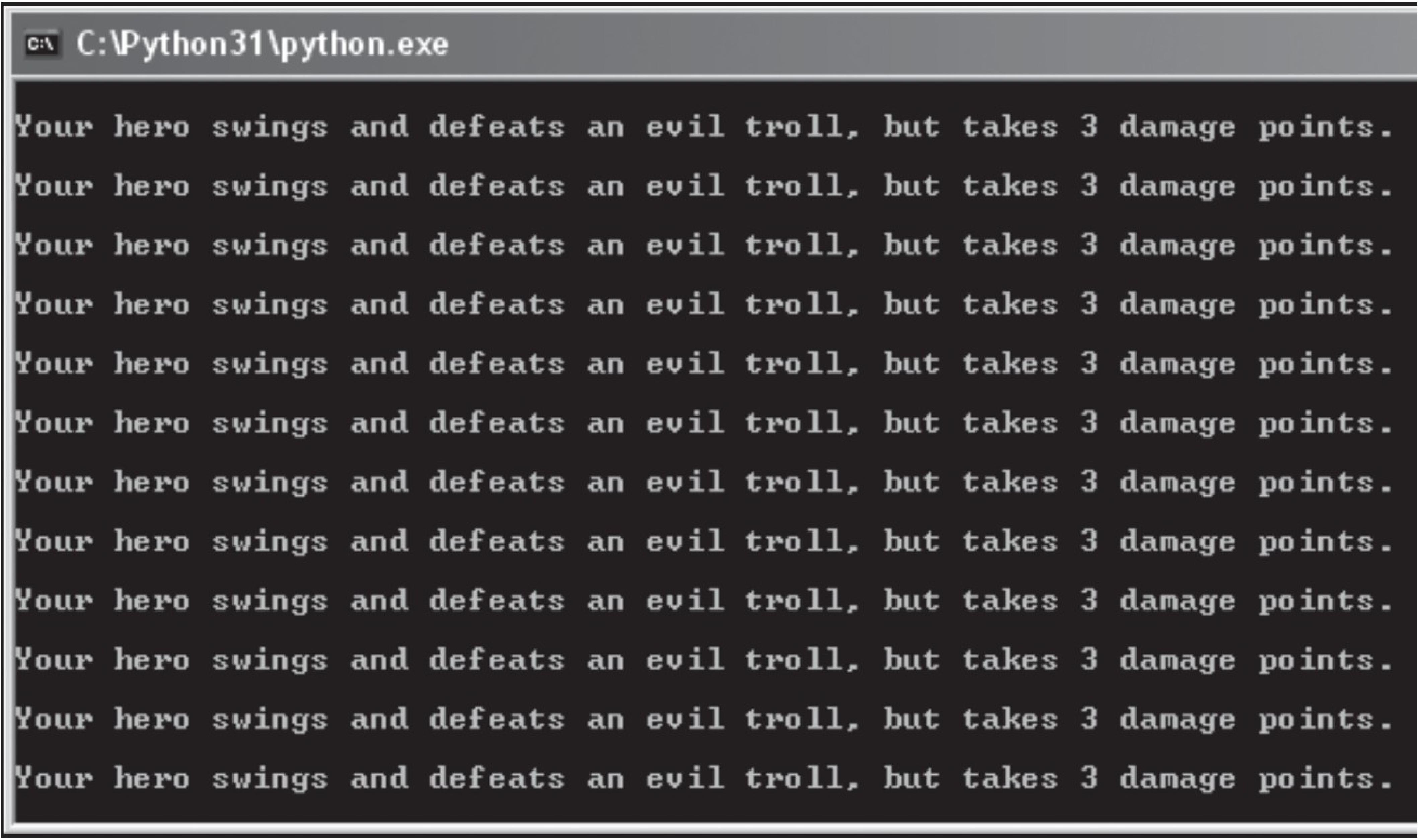
- If you write a loop that never stops, you’ve created an *infinite* loop.

- a simple example of an infinite loop:

```
counter = 0
while counter <= 10:
      print(counter)
```

- The values in the condition must be able to change as a result of the code inside the loop body. If they can’t ever change, the loop won’t end, and you end up with an infinite loop

Introducing the Losing Battle Program



```
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
Your hero swings and defeats an evil troll, but takes 3 damage points.
```

- Stop the process by pressing Ctrl+C, or it runs forever.

losing_battle-bad.py

```
# Losing Battle
# Demonstrates the dreaded infinite loop

print("Your lone hero is surrounded by a massive “,
          “army of trolls.")
print("Their decaying green bodies stretch out,”,
          “melting into the horizon.")
print("Your hero unsheathes his sword for the last “,
          “fight of his life.\n")

health = 10
trolls = 0
damage = 3
```

```
while health != 0:
    trolls += 1
    health -= damage

    print("Your hero swings and defeats an “,
             “evil troll, but takes", damage,
             "damage points.\n")

print("Your hero fought valiantly and defeated",
          trolls, "trolls.")
print("But alas, your hero is no more.")

input("\n\nPress the enter key to exit.")
```

Tracing the Program

- It looks like the program has a logical error. A good way to track down the error is to trace your program’s execution.

- *Tracing* means you simulate the running of your program and do exactly what it would do, following every statement and keeping track of the values assigned to variables.

- This way, you can step through the program, understand exactly what is happening at each point, and discover the circumstances that conspire to produce the bug in your code.

- After a few times through the loop, the trace looks like:

health	trolls	damage	health != 0
10	0	3	True
7	1	3	True
4	2	3	True
1	3	3	True
-2	4	3	True
-5	5	3	True
-7	6	3	True

- Since the value of health is negative (and not equal to 0) in the last 3 lines of the trace, the condition is still True .

- health will never become 0. It will just grow in the negative direction each time the loop executes. As a result, the condition will never become False , and the loop will never end.

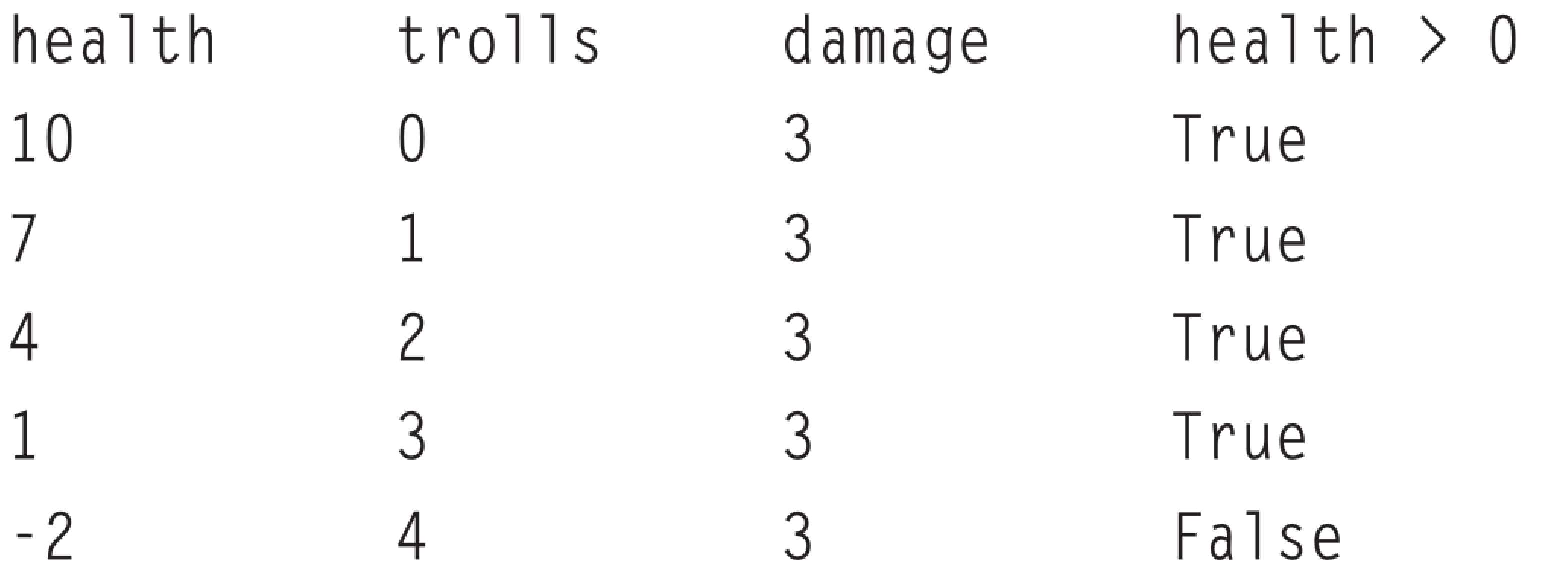
Creating Condition That Can Become False

- Besides making sure values in a while loop’s condition change, you should be sure that the condition can eventually evaluate to False ; otherwise, you still have an infinite loop.

- the fix to the previous program is easy. The line with the condition just needs to become

while health > 0:

- Now, if health becomes 0 or negative, the condition evaluates to False and the loop ends.

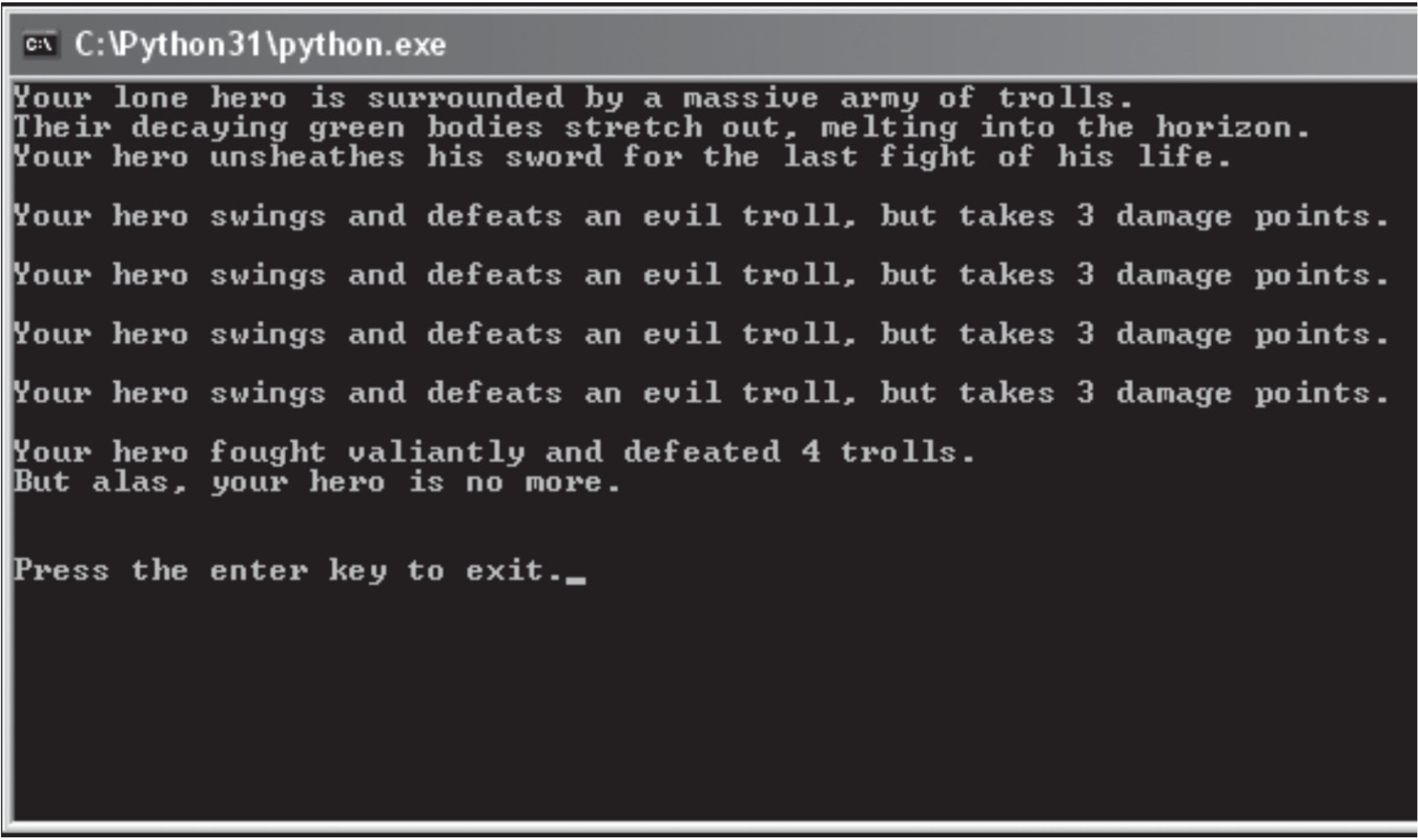


health	trolls	damage	health > 0
10	0	3	True
7	1	3	True
4	2	3	True
1	3	3	True
-2	4	3	False

losing_battle-good.py

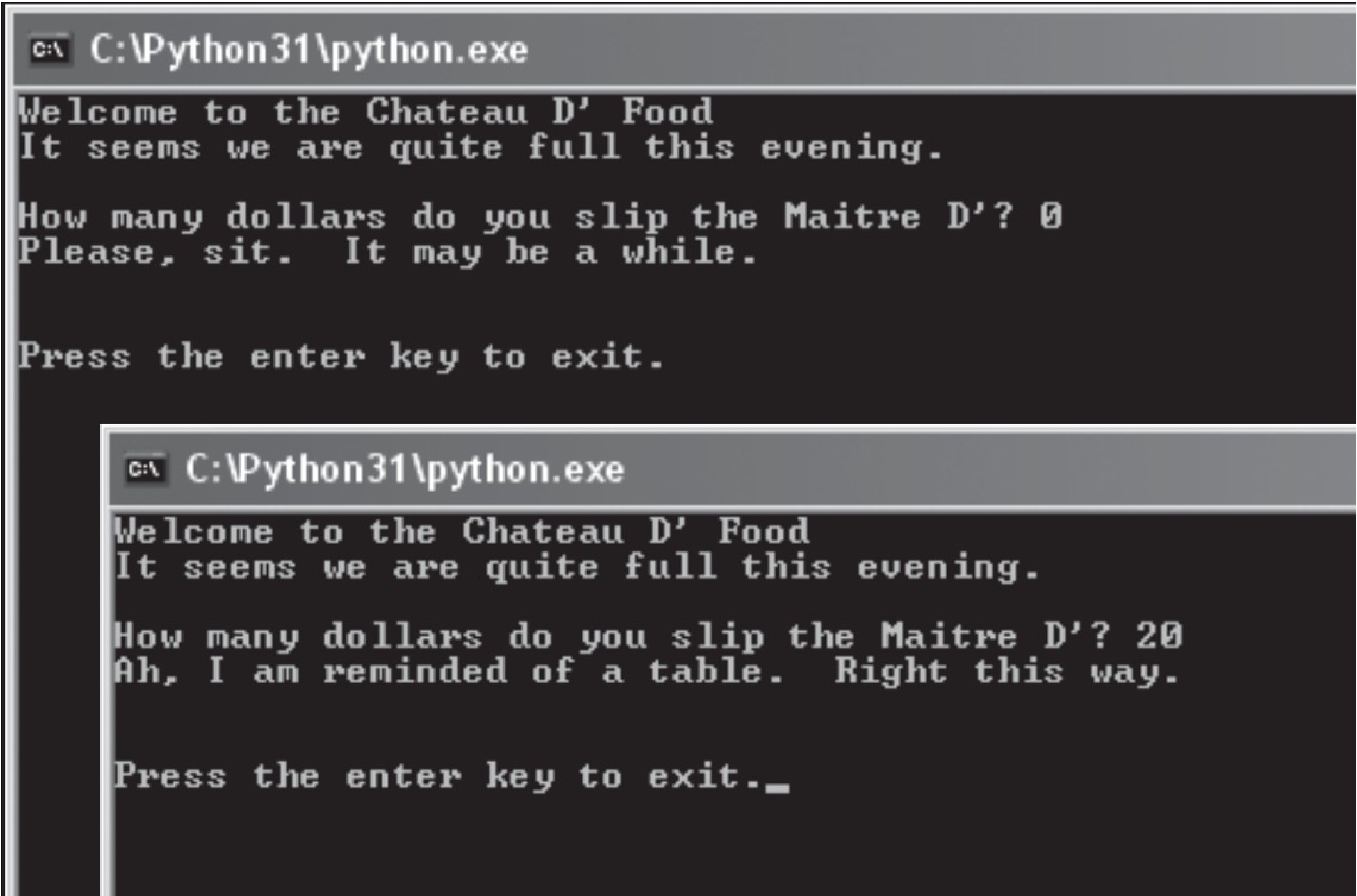
```
# Losing Battle
# Demonstrates the dreaded infinite loop

print("Your lone hero is surrounded by a massive “,
        “army of trolls.")
print("Their decaying green bodies stretch out,”,
        “melting into the horizon.")
print("Your hero unsheathes his sword for the last “,
        “fight of his life.\n")

health = 10
trolls = 0
damage = 3
```

```
while health > 0:
    trolls += 1
    health -= damage

    print("Your hero swings and defeats an “,
            “evil troll, but takes", damage,
            "damage points.\n")

print("Your hero fought valiantly and defeated",
        trolls, "trolls.")
print("But alas, your hero is no more.")

input("\n\nPress the enter key to exit.")
```


C:\Python31\python.exe

```
Your lone hero is surrounded by a massive army of trolls.
Their decaying green bodies stretch out, melting into the horizon.
Your hero unsheathes his sword for the last fight of his life.

Your hero swings and defeats an evil troll, but takes 3 damage points.

Your hero swings and defeats an evil troll, but takes 3 damage points.

Your hero swings and defeats an evil troll, but takes 3 damage points.

Your hero swings and defeats an evil troll, but takes 3 damage points.

Your hero fought valiantly and defeated 4 trolls.
But alas, your hero is no more.


Press the enter key to exit._
```

Introducing the Maitre D' Program



```
Welcome to the Chateau D' Food
It seems we are quite full this evening.

How many dollars do you slip the Maitre D'? 0
Please, sit.  It may be a while.


Press the enter key to exit.
```

```
Welcome to the Chateau D' Food
It seems we are quite full this evening.

How many dollars do you slip the Maitre D'? 20
Ah, I am reminded of a table.  Right this way.


Press the enter key to exit._
```

maitre_d.py

```
# Maitre D'
# Demonstrates treating a value as a condition

print("Welcome to the Chateau D' Food")
print("It seems we are quite full this evening.\n")

money = int(input("How many dollars do you slip the
Maitre D'? "))

if money:
    print("Ah, I am reminded of a table. “,
              ”Right this way.")
else:
    print("Please, sit.  It may be a while.")

input("\n\nPress the enter key to exit.")
```

Interpreting Any Value as True or False

- The new concept is demonstrated in the line:

if money:

- money is not compared to any other value. money is the condition.

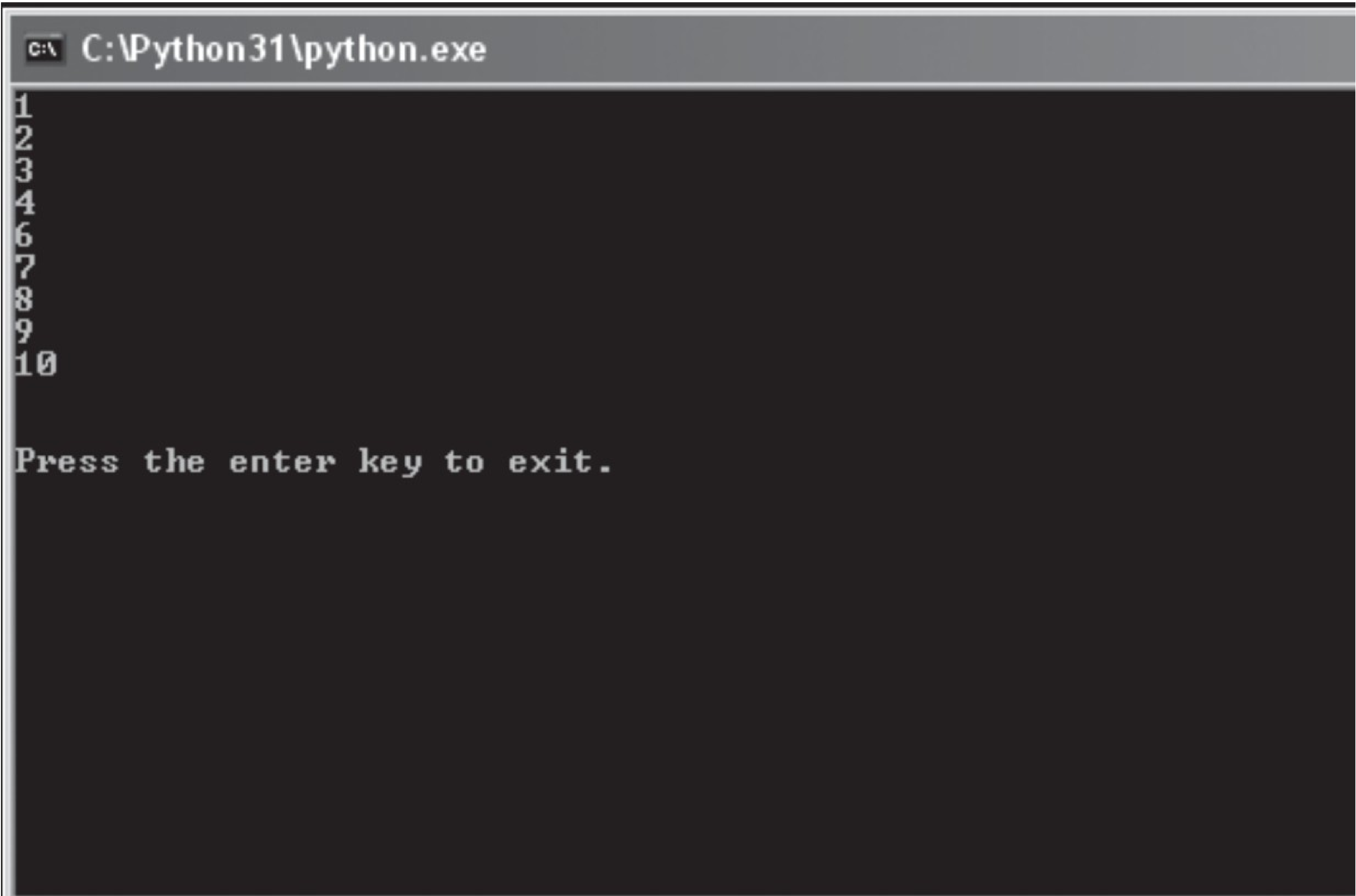
- When it comes to evaluating a number as a condition, **0** is False and **everything else** , even if it is negative, is True .

- So, the above line is equivalent to if money != 0:

- The basic principle is: any empty or zero value is False , everything else is True .

- The empty string, "" , is False , any other string is True .

Introducing the Finicky Counter Program



C:\Python31\python.exe

```
1
2
3
4
6
7
8
9
10


Press the enter key to exit.
```

finicky_counter.py

```
# Finicky Counter
# Demonstrates the break and continue statements

count = 0
while True:
    count += 1
    # end loop if count greater than 10
    if count > 10:
        break
    # skip 5
    if count == 5:
        continue
    print(count)

input("\n\nPress the enter key to exit.")
```

Using the break Statement to Exit a Loop

- The loop is set up with:

```
while True:
```

- This technically means that the loop will continue forever, unless there is an exit condition in the loop body. So we put

```
# end loop if count greater than 10
if count > 10:
   break
```

- Since count is increased by 1 each time the loop body begins, it will eventually reach 11.

- When it does, the **break** statement, which here means “break out of the loop,” is executed and the loop ends.

Using the continue Statement to Jump Back to the Top of a Loop

- For the lines:

```
# skip 5
if count == 5:
    continue
```

- The **continue** statement means “jump back to the top of the loop.” At the top of the loop, the while condition is tested and the loop is entered again if it evaluates to True .

- So when count is equal to 5, the program does not get to the print(count) statement. Instead, it goes right back to the top of the loop so that 5 is never printed.

When to Use break and continue

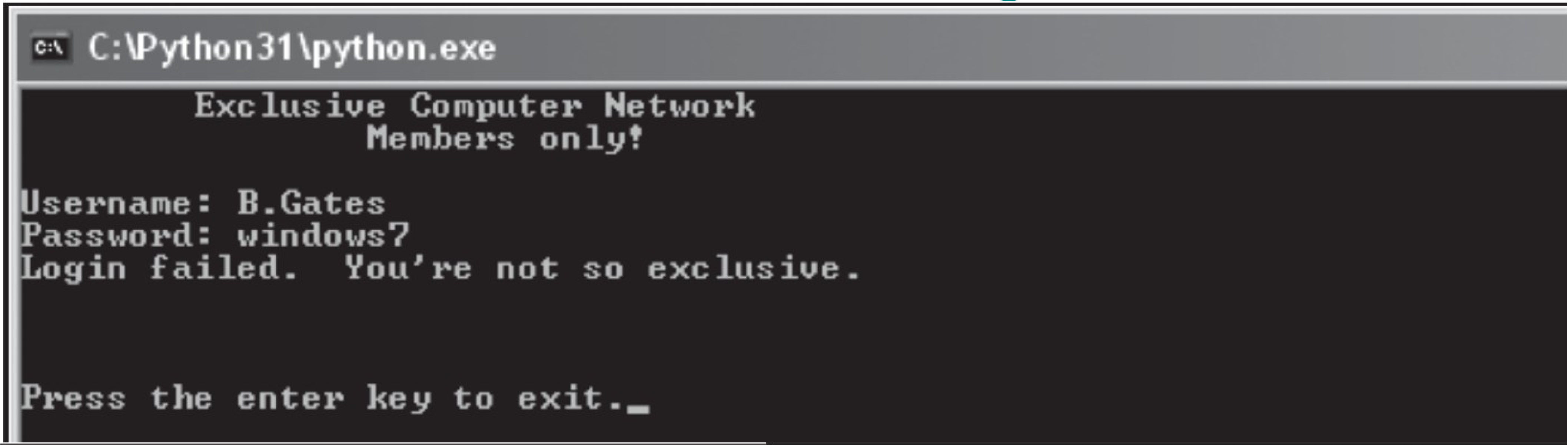
- You can use break and continue in any loop you create.

- But both break and continue make it harder to see the flow of a loop and understand under what conditions it ends.

- You don’t actually need break and continue .

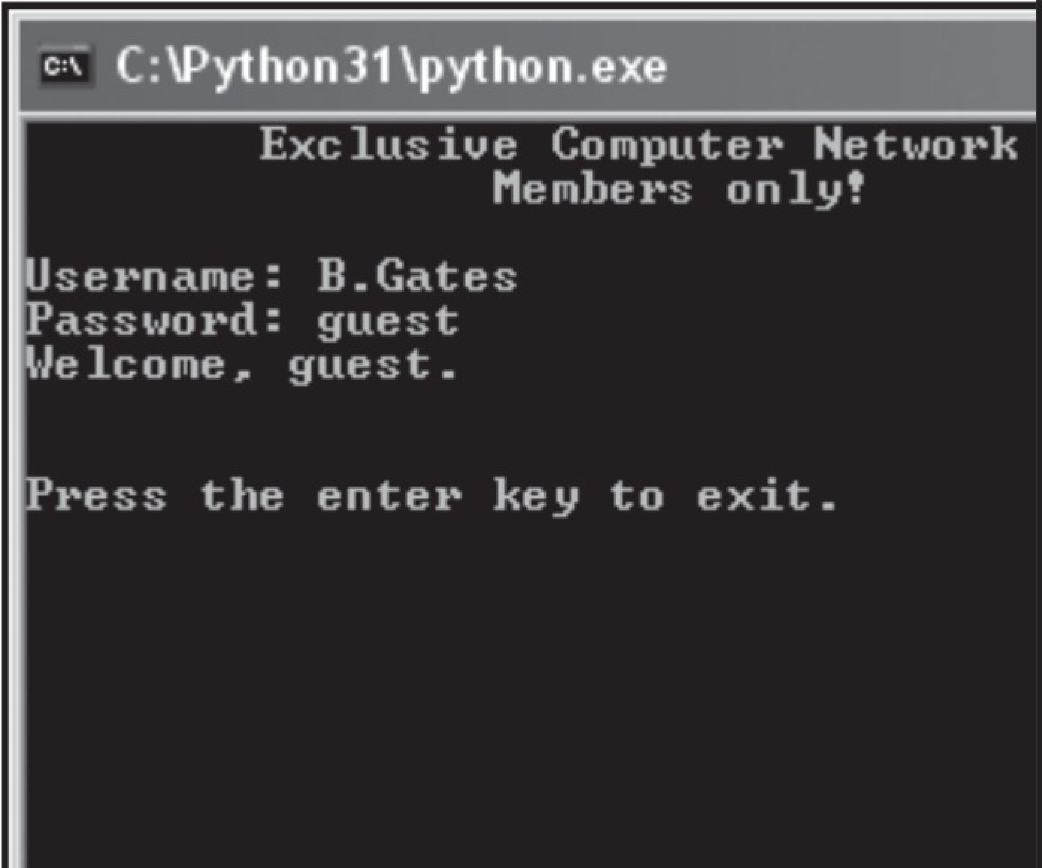
- Any loop you can write using them can be written without them (in most of the cases).

The Exclusive Network Program



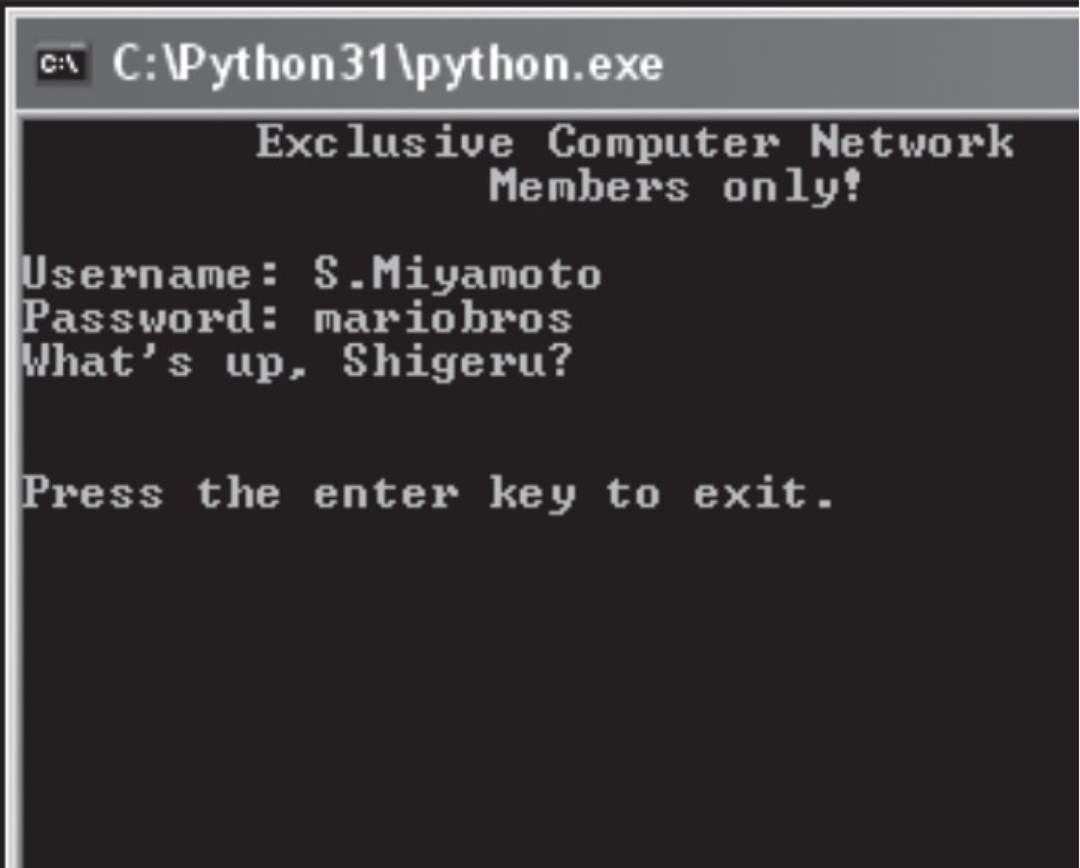
C:\Python31\python.exe

```
        Exclusive Computer Network
                Members only!

Username: B.Gates
Password: windows7
Login failed.  You're not so exclusive.



Press the enter key to exit._
```



C:\Python31\python.exe

```
        Exclusive Computer Network
                Members only!

Username: B.Gates
Password: guest
Welcome, guest.


Press the enter key to exit.
```



C:\Python31\python.exe

```
        Exclusive Computer Network
                Members only!

Username: S.Miyamoto
Password: mariobros
What's up, Shigeru?


Press the enter key to exit.
```

exclusive_network.py

```python
# Exclusive Network
# Demonstrates logical operators and compound conditions

print("\tExclusive Computer Network")
print("\t\tMembers only!\n")

security = 0

username = ""
while not username:
    username = input("Username: ")

password = ""
while not password:
    password = input("Password: ")
```

```
if username == "M.Dawson" and password == "secret":
    print("Hi, Mike.")
    security = 5
elif username == "S.Meier" and password == "civilization":
    print("Hey, Sid.")
    security = 3
elif username == "S.Miyamoto" and password == "mariobros":
    print("What's up, Shigeru?")
    security = 3
elif username == "W.Wright" and password == "thesims":
    print("How goes it, Will?")
    security = 3
elif username == "guest" or password == "guest":
    print("Welcome, guest.")
    security = 1
else:
    print("Login failed.  You're not so exclusive.\n")

input("\n\nPress the enter key to exit.")
```

Understanding the not Logical Operator

- Want to make sure that the user enters something for the username and password. Just pressing the Enter key, which results in the empty string, won't do.

- Want a loop that continues to ask for a username until the user enters something:

```
username = ""
while not username:
      username = input("Username: ")
```

- The logical **not** operator works a lot like the word "not." In Python, putting **not** in front of a condition creates a new condition that evaluates to the opposite of the original.

username	not username
True	False
False	True

- Since username is initialized to the empty string in the program, it starts out as False . That makes not username True and the loop runs the 1st time.

- If the user just presses Enter, username is the empty string, just as before. And not username is True and the loop keeps running.

- When the user finally enters something, username becomes something other than the empty string. That makes username evaluate to True and not username evaluate to False . As a result, the loop ends.

- The program does the same thing for password .

Understanding the and Logical Operator

- A user has to enter a username and password that are recognized together. The program checks that a user enters S.Meier for his username, civilization for his password with

```
elif username == "S.Meier" and password == "civilization":
```

- The line contains a single compound condition made up of 2 simple conditions. They are joined together by the **and** logical operator to form a larger, compound condition.

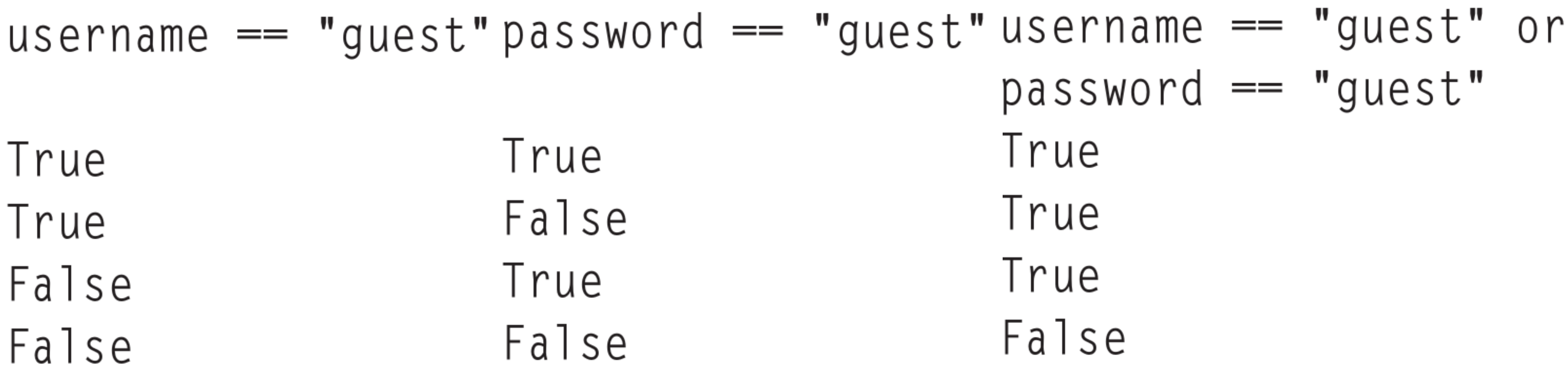
- This compound condition, though larger, is still just a condition, which means that it can be either True or False .

username == "S.Meier"	password == "civilization"	username == "S.Meier" and password == "civilization
True	True	True
True	False	False
False	True	False
False	False	False

Understanding the or Logical Operator

- Guests are allowed in the network, but with a limited security level. To make it easy, all a guest has to do is enter guest for either the username or password:

```
elif username == "guest" or password == "guest":
    print("Welcome, guest.")
    security = 1
```



username == "guest"	password == "guest"	username == "guest" or password == "guest"
True	True	True
True	False	True
False	True	True
False	False	False

Planning Your Programs: Creating Algorithms with Pseudocode

- An *algorithm* is a set of clear, easy-to-follow instructions for accomplishing some task. An algorithm is like an outline for your program. It’s something you planned out, before programming, to guide you along as you code.

- An algorithm isn’t just a goal—it’s a concrete list of steps to be followed in order.

- Algorithms are generally written in *pseudocode*.

- Pseudocode falls somewhere between English and a programming language.

- Make a Million Dollars algorithm:

```
if you can think of a new and useful product
    then that’s your product
otherwise
    repackage an existing product as your product

make an infomercial about your product

show the infomercial on TV

charge $100 per unit of your product

sell 10,000 units of your product
```

Stepwise Refinement to Your Algorithms

- Often, algorithms need multiple passes before they can be implemented in code.
- *Stepwise refinement* is one process used to rewrite algorithms so that they're ready for implementation.
- By taking each step in an algorithm and breaking it down into a series of simpler steps, the algorithm becomes closer to programming code.
- In stepwise refinement, you keep breaking down each step until you feel that the entire algorithm could be fairly easily translated into a program.
- If you're still unclear about a step, refine it some more. Continue with this process and you will have a complete algorithm.

- As an example,

create an infomercial about your product

- This might seem like too vague a task. Using stepwise refinement, the single step can be broken down into several others:

write a script for an infomercial about your product

rent a TV studio for a day

hire a production crew

hire an enthusiastic audience

film the infomercial

Guess My Number: Planning the Program

- To plan the game, write some pseudocode first:

```
pick a random number

while the player hasn't guessed the number
    let the player guess

congratulate the player
```

- It is missing some important elements:

1. the program needs to tell the player if the guess is too high or too low.

2. the program should keep track of how many guesses the player has made and then tell the player this number at the end of the game.

- A refinement of the algorithm:

```
welcome the player to the game and explain it

  pick a random number between 1 and 100
  ask the player for a guess
  set the number of guesses to 1

       while the player's guess does not equal the number
              if the guess is greater than the number
                   tell the player to guess lower
              otherwise
                   tell the player to guess higher
              get a new guess from the player
              increase the number of guesses by 1

congratulate the player on guessing the number
let the player know how many guesses it took
```

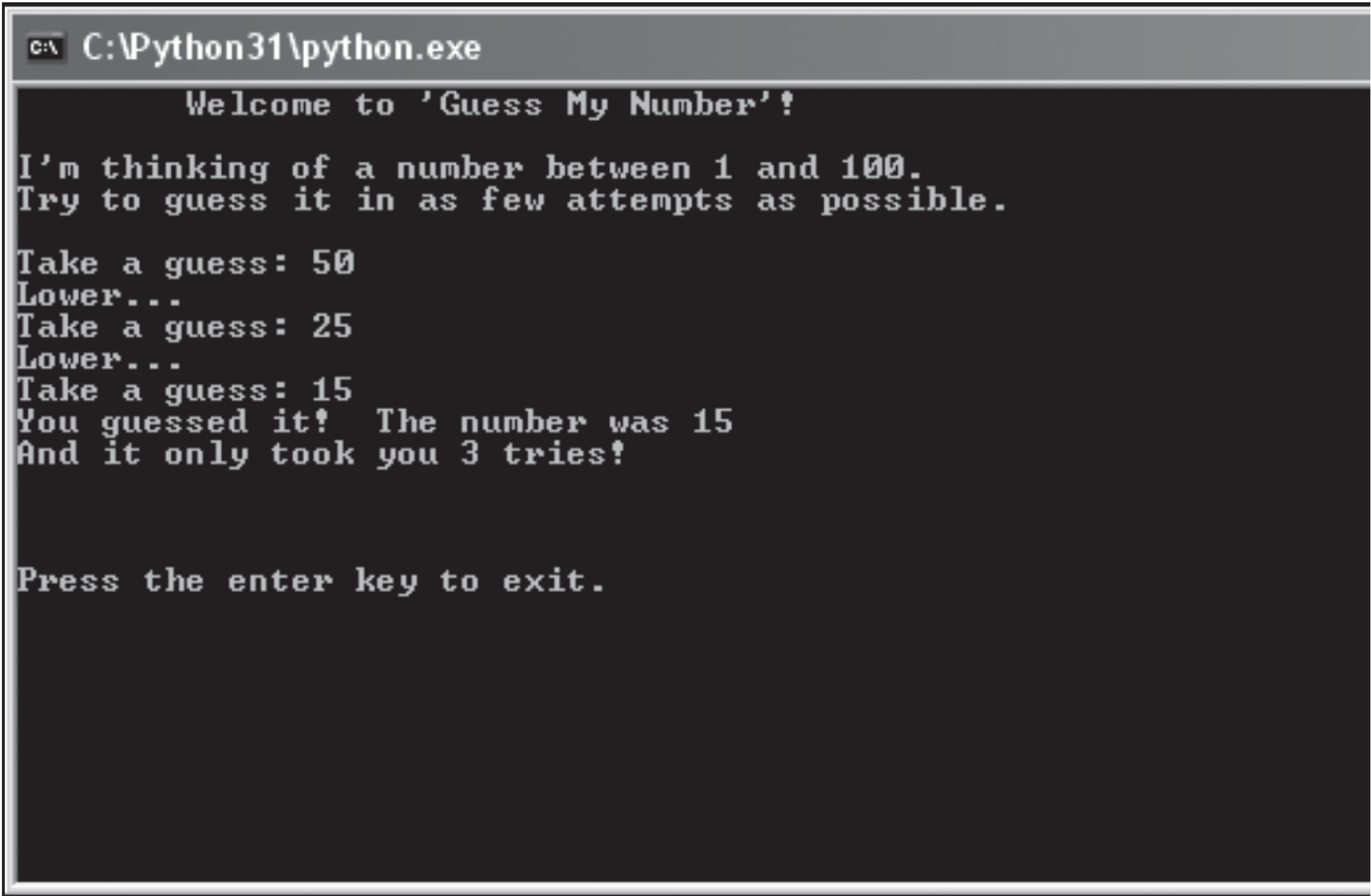
- Much better! Ready to go!

guess_my_ number.py

```
# Guess My Number
#
# The computer picks a random number between 1-100
# The player tries to guess it and the computer lets
# the player know if the guess is too high, too low
# or right on the money

import random

print("\tWelcome to 'Guess My Number'!")
print("\nI'm thinking of a number between 1 and 100.")
print("Try to guess it in as few attempts as
possible.\n")

# set the initial values
the_number = random.randint(1, 100)
guess = int(input("Take a guess: "))
tries = 1
```

```
# guessing loop
while guess != the_number:
    if guess > the_number:
        print("Lower...")
    else:
        print("Higher...")

    guess = int(input("Take a guess: "))
    tries += 1

print("You guessed it!  The number was", the_number)
print("And it only took you", tries, "tries!\n")

input("\n\nPress the enter key to exit.")
```

the Result of the Guess My Number Game



```
        Welcome to 'Guess My Number'!

I'm thinking of a number between 1 and 100.
Try to guess it in as few attempts as possible.

Take a guess: 50
Lower...
Take a guess: 25
Lower...
Take a guess: 15
You guessed it!  The number was 15
And it only took you 3 tries!



Press the enter key to exit.
```

Quiz 3: Write the pseudocode for a program where the player and the computer trade places in the number guessing game. That is, the player picks a random number between 1 and 100 that the computer has to guess. Then code the game according to your pseudocode.