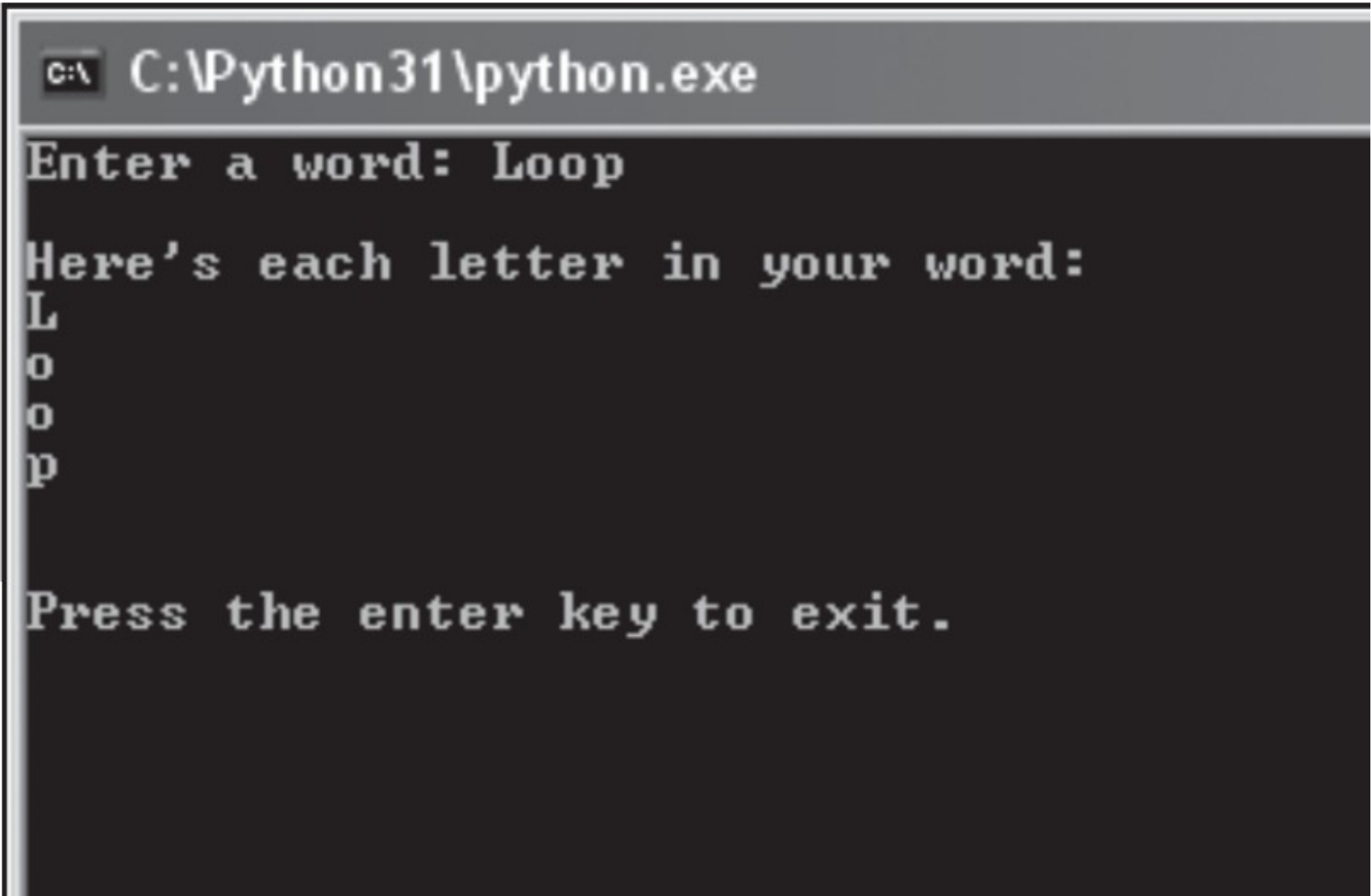
Chapter 4

For Loops, Strings, Tuples: The Word Jumble Game

Introducing the Loopy String Program



loopy_string.py

```
# Loopy String
# Demonstrates the for loop with a string

word = input("Enter a word: ")

print("\nHere's each letter in your word:")

for letter in word:
    print(letter)

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

Understanding for Loops

- A **for** loop repeats its loop body for each *element* of the sequence, in order. It marches through (or *iterates over*) a sequence one element at a time.

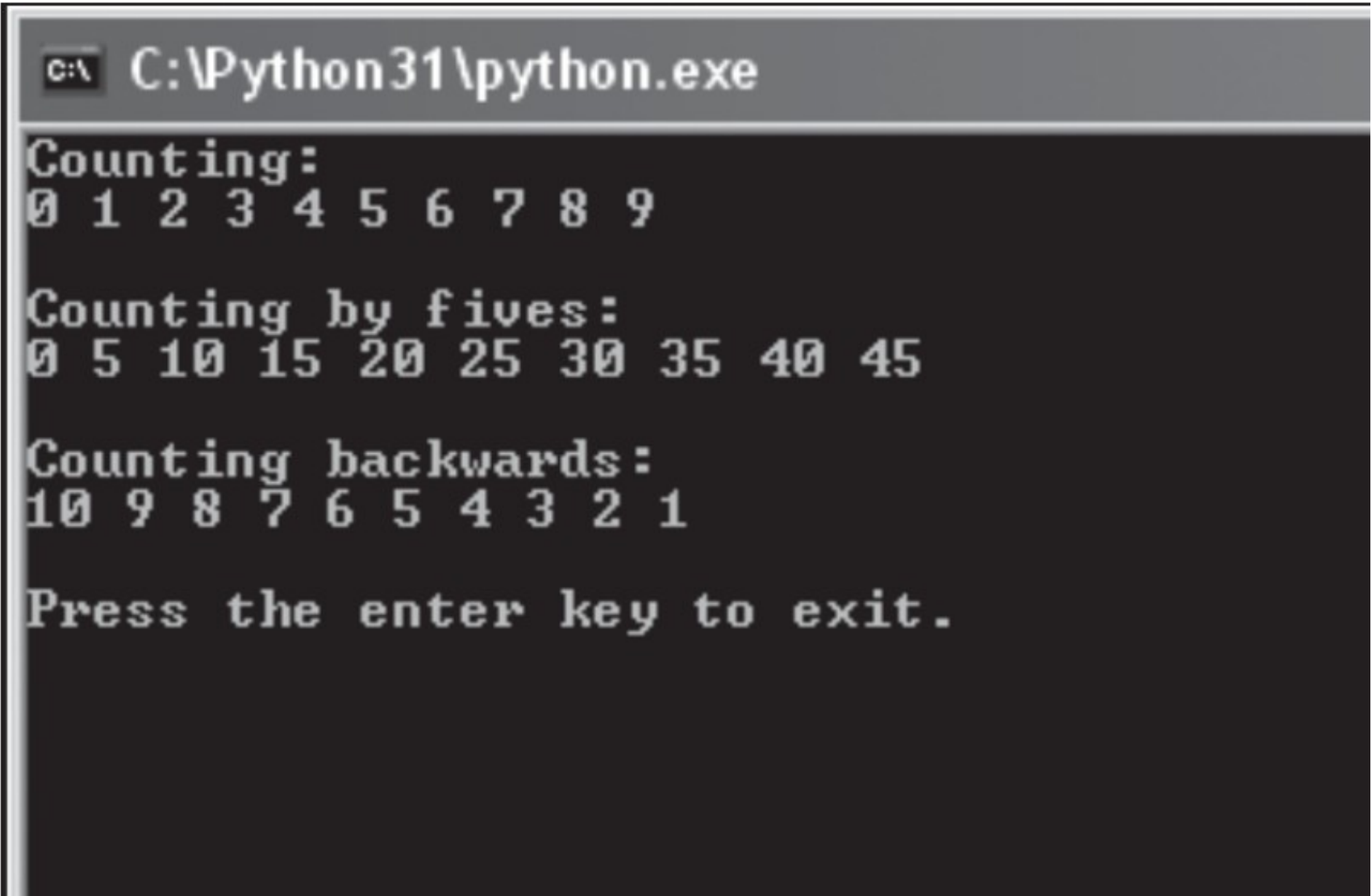
- The **for** loop is as follows:

```
for letter in word:
    print(letter)
```

- In the case of the string "Loop" , the 1st element is the character "L" , the 2nd is "o" , and so on.

- A **for** loop uses a variable that gets each successive element of the sequence, eg, letter .

Introducing the Counter Program



counter.py

```
# Counter
# Demonstrates the range() function

print("Counting:")
for i in range(10):
    print(i, end=" ")

print("\n\nCounting by fives:")
for i in range(0, 50, 5):
    print(i, end=" ")

print("\n\nCounting backwards:")
for i in range(10, 0, -1):
    print(i, end=" ")

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

Counting Forwards

- The 1st loop in the program counts forwards:

```
for i in range(10):
    print(i, end=" ")
```

- The sequence the loop iterates over is generated by the return value of the **range()** function.

- If you give **range()** a positive integer, you can imagine that it returns a sequence starting with 0, up to, but not including, the number you gave it.

- **range(10)** returns the sequence [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

- To be more formal, **range(10)** = **range(0,10,1)** .

Counting by Fives

- The next loop counts by fives:

```
for i in range(0, 50, 5):
    print(i, end=" ")
```

- If you give range() 3 values, it will treat them as a start point, an end point, and the number by which to count.

- The start point is always the 1st value in our imagined sequence while the end point is never included.

- So the sequence is [0, 5, 10, 15, 20, 25, 30, 35, 40, 45] .

- Notice that the sequence ends at 45, not 50.

- If you want to include 50, your end point needs to be greater than 50, eg, range(**0, 51, 5**).

Counting Backwards

- The last loop in the program counts backwards:

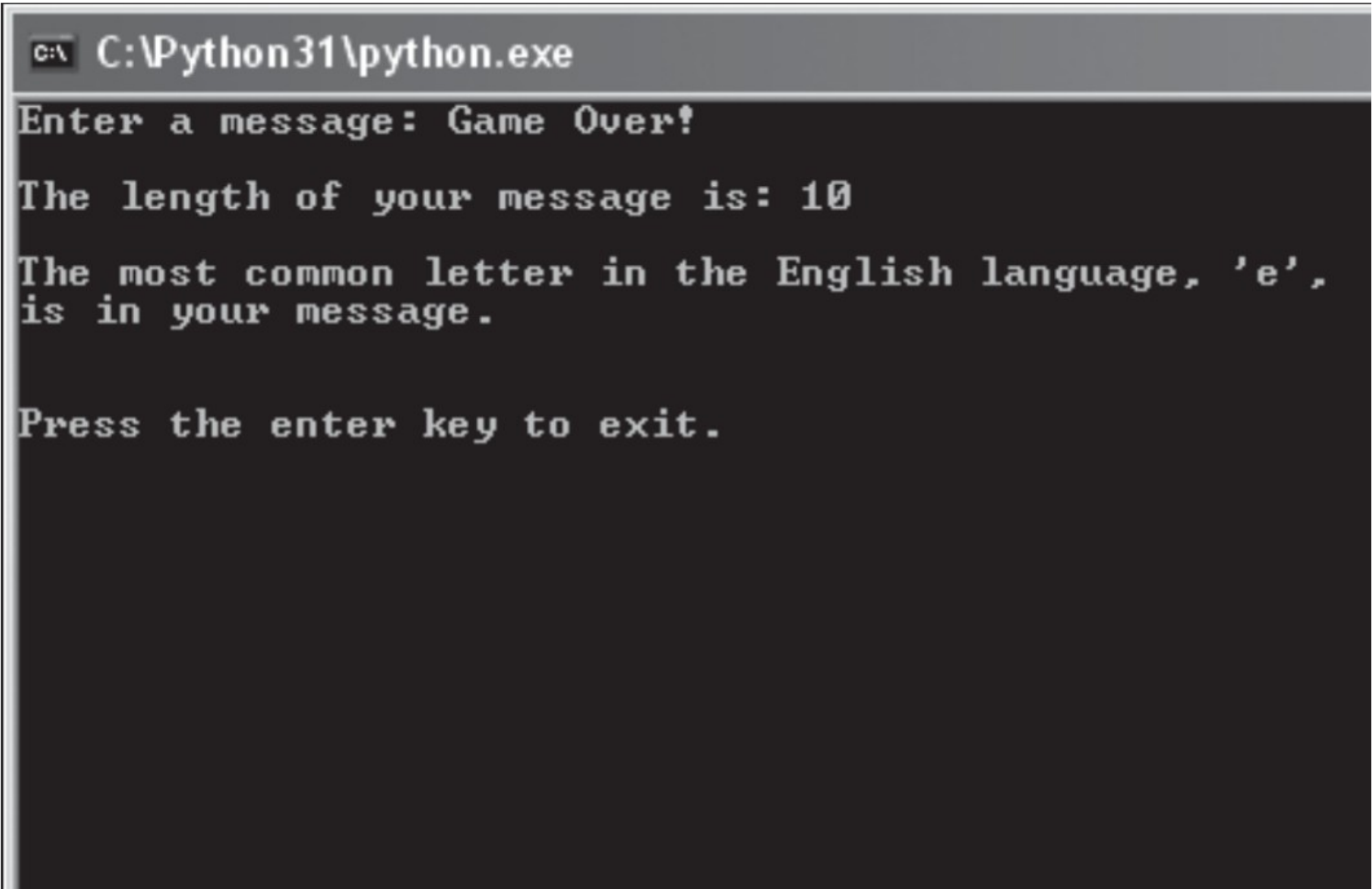
```
for i in range(10, 0, -1):
    print(i, end=" ")
```

- Notice that the last argument in the range() call is –1. This tells the function to go from the start point to the end point by adding –1 each time.

- So the sequence is [10, 9, 8, 7, 6, 5, 4, 3, 2, 1] .

- The loop counts from 10 down to 1 and does not include 0.

The Message Analyzer Program



message_analyzer.py

```
# Message Analyzer
# Demonstrates the len() function and the in operator

message = input("Enter a message: ")

print("\nThe length of your message is:", len(message))

print("\nThe most common letter in the English
language, 'e',")

if "e" in message:
    print("is in your message.")
else:
    print("is not in your message.")

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

Using the len() Function

- You can pass any sequence you want to the **len()** function and it will return the length of the sequence:

```
print("\nThe length of your message is:", len(message))
```

- A sequence's length is the number of elements it has.

Using the in Operator

- The program uses the following lines to test whether "e" is in the message the user entered:

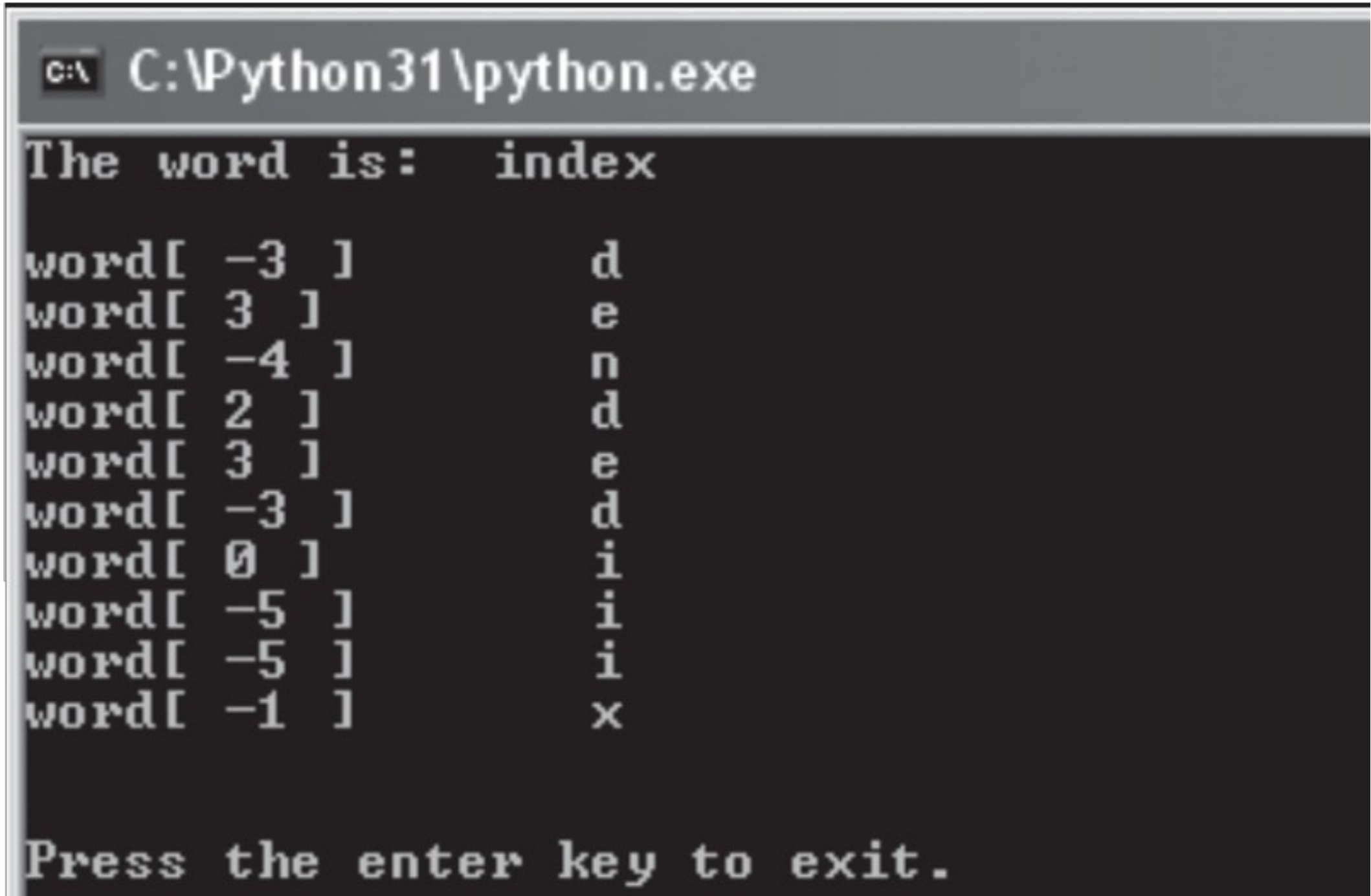
```
if "e" in message:
    print("is in your message.")
else:
    print("is not in your message.")
```

- If message contains the character "e", it's true. If message doesn't contain "e", it's false.

- the value of message is "Game Over!". So, the condition "e" in message evaluated to True and the computer printed "is in your message."

- You can use **in** anywhere to check if an element is a member of a sequence. If the element is a member, the condition is true; otherwise, it's false.

Introducing the Random Access Program



random_access.py

```
# Random Access
# Demonstrates string indexing

import random

word = "index"
print("The word is: ", word, "\n")

high = len(word)
low = - len(word)

for i in range(10):
    position = random.randrange(low, high)
    print("word[", position, "]\t", word[position])

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

Working with Positive Position Numbers

- For the string variable word = "index", the 1st letter, “i,” is at position **0**. The 2nd letter, “n,” is at position **1**. The 3rd letter, “d,” is at position **2**, and so on.

- To access the letter in position 0 from the variable word , you’d just type **word[0]** . For any other position, you’d just substitute that number.

```
>>> word = "index "
>>> print(word[0])
i
>>> print(word[1])
n
>>> print(word[2])
d
>>> print(word[3])
e
>>> print(word[4])
x
```

- There is no position 5 in this string, because the computer begins counting at 0. Valid positive positions are 0, 1, 2, 3, 4.

- Any attempt to access a position 5 will cause an error:

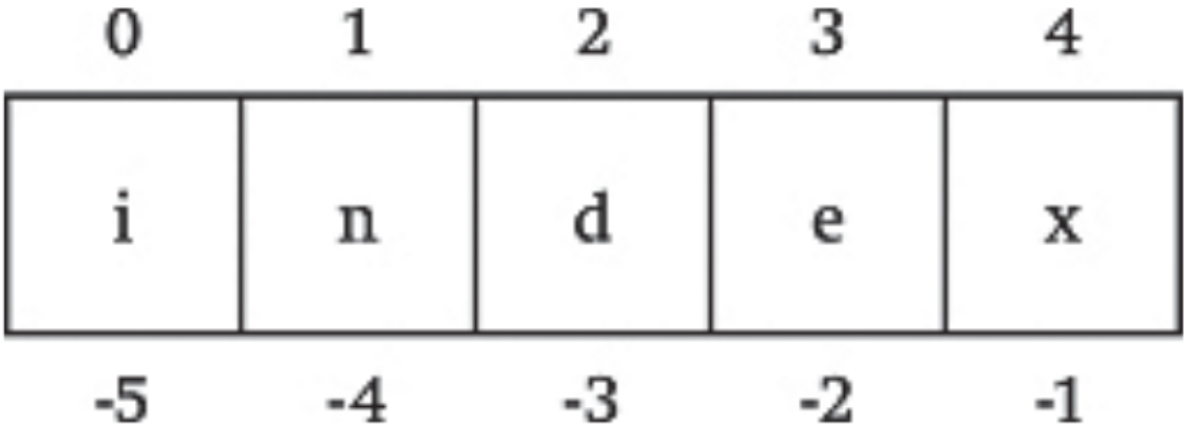
```
>>> word = "index "
>>> print(word[5])
Traceback (most recent call last):
          File "<pyshell#1>", line 1, in ? print word[5]
IndexError: string index out of range
```

Working with Negative Position Numbers

- There's also a way to access elements of a sequence through negative position numbers.

- With negative position numbers, you start counting from the end. For strings, that means you start counting from the last letter and work backwards.

```
>>> word = "index "
>>> print(word[-1])
x
>>> print(word[-2])
e
>>> print(word[-3])
d
>>> print(word[-4])
n
>>> print(word[-5])
i
```



Accessing a Random String Element

- To access a random letter from the "index", the 1st thing is to import the random module:

import random

- Then generate a random number between –5 and 4, because those are all the possible position values of word .

- The random.randrange() function can produce a random number from between 2 numbers:

high = len(word)
low = - len(word)

position = random.randrange(low, high)

produces either –5, –4, –3, –2, –1, 0, 1, 2, 3, 4, but not 5.

- Finally, create a for loop that executes 10 times to picks a random position value and prints that position value and corresponding letter:

```
for i in range(10):
    position = random.randrange(low, high)
    print("word[", position, "]\t", word[position])
```

Understanding String Immutability

- Sequences fall into one of two categories: mutable or immutable. *Mutable* means changeable. *Immutable* means unchangeable.

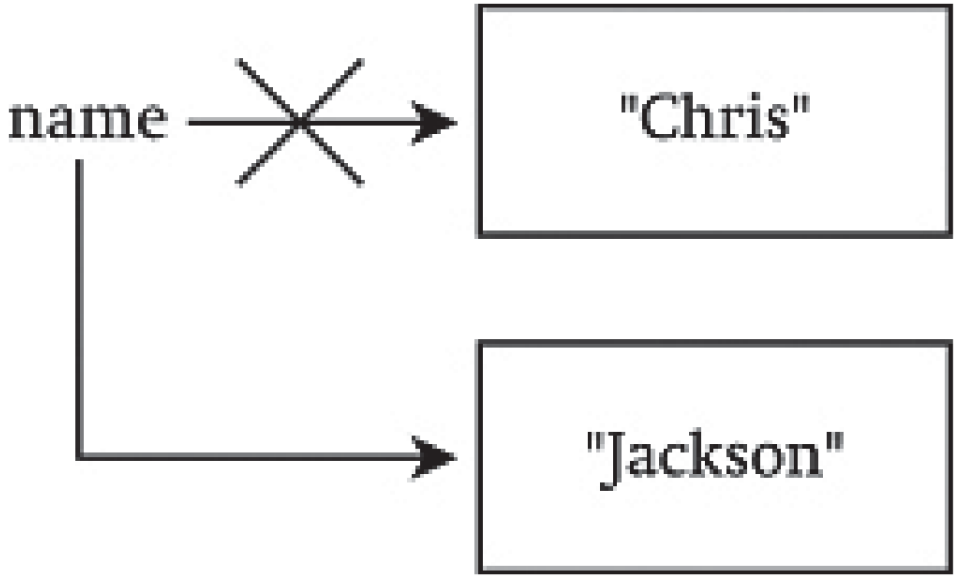
- Strings are immutable sequences, which means that they can’t change.

- For example, the string "Game Over!" will always be the string "Game Over!" . You can’t change it. In fact, you can’t change any string you create.

```
>>> name = "Chris "
>>> print(name)
Chris
>>> name = "Jackson "
>>> print(name)
Jackson
```

- In this case you might think that you can change a string. But, you didn’t change any strings in this session.

- It is only a reassignment of a variable to different string.

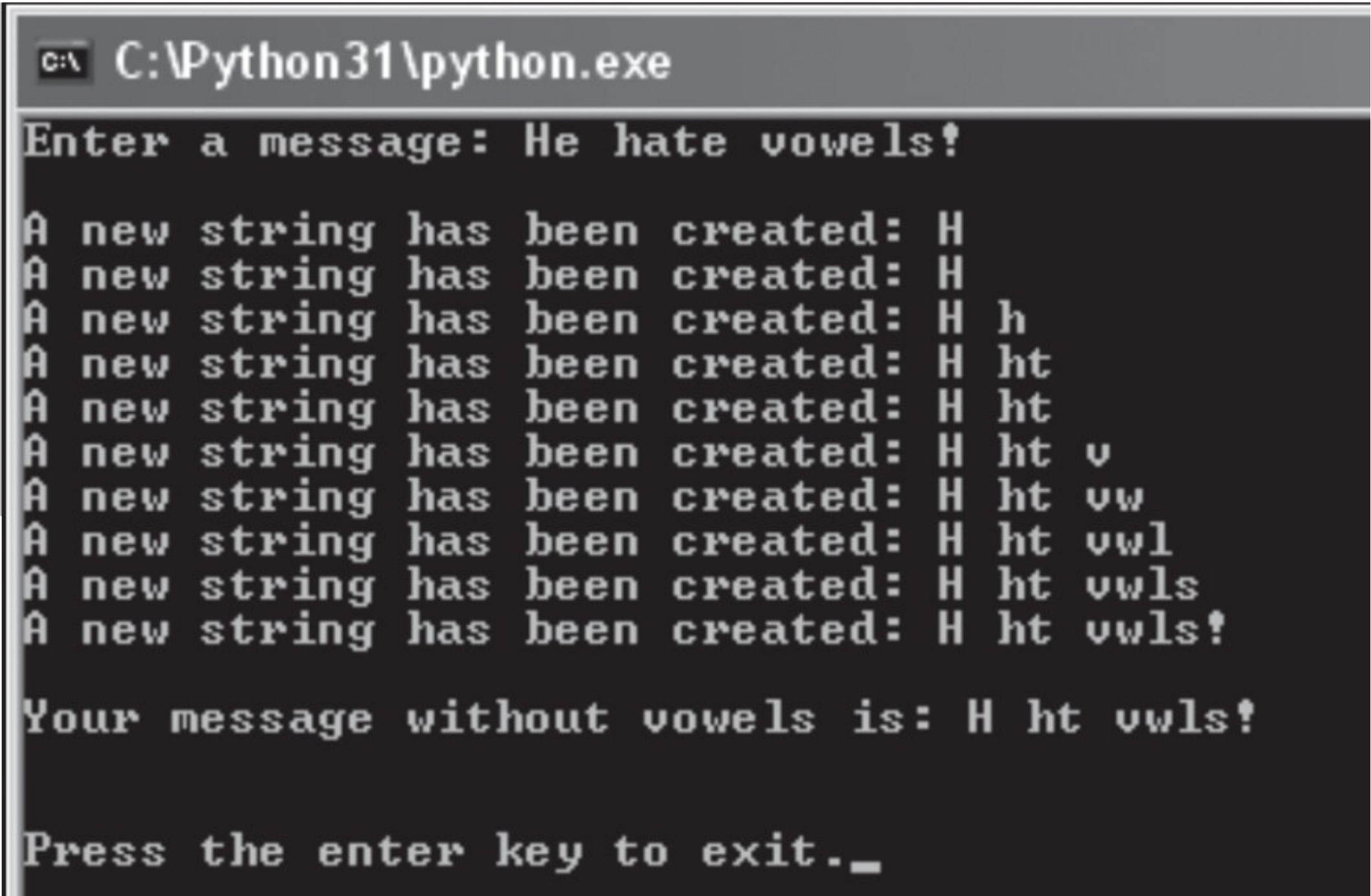


- Since you can’t change a string, you can’t assign a new character to a string through indexing.

```
>>> word = "game "
>>> word[0] = "l "
Traceback (most recent call last):
    File "<pyshell#1>", line 1, in <module> word[0] = "l"
TypeError: 'str' object does not support item assignment
```

- You can’t alter a string, but you can create new strings from existing ones.

Introducing the No Vowels Program



no_vowels.py

```
# No Vowels
# Demonstrates creating new strings with a for loop

message = input("Enter a message: ")
new_message = ""
VOWELS = "aeiou"

print()
for letter in message:
    if letter.lower() not in VOWELS:
        new_message += letter
        print("A new string has been created:",
        new_message)

print("\nYour message without vowels is:", new_message)

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

Creating Constants

- Traditionally, variable names are in lowercase.

- There's a special meaning associated with variable names in all caps. They're called *constants* and refer to a value that is not meant to change (their value is constant):

VOWELS = "aeiou"

- Constants are valuable to programmers in 2 ways:

1. they make programs clearer.
2. constants save retyping (and possibly errors in typing).

- There's nothing in Python that will stop you from changing a "constant" in your program. This naming practice is simply a convention.

Creating New Strings from Existing Ones

- The program can't literally add a character to a string, so, it concatenates the new message it has so far with a character to create a new string:

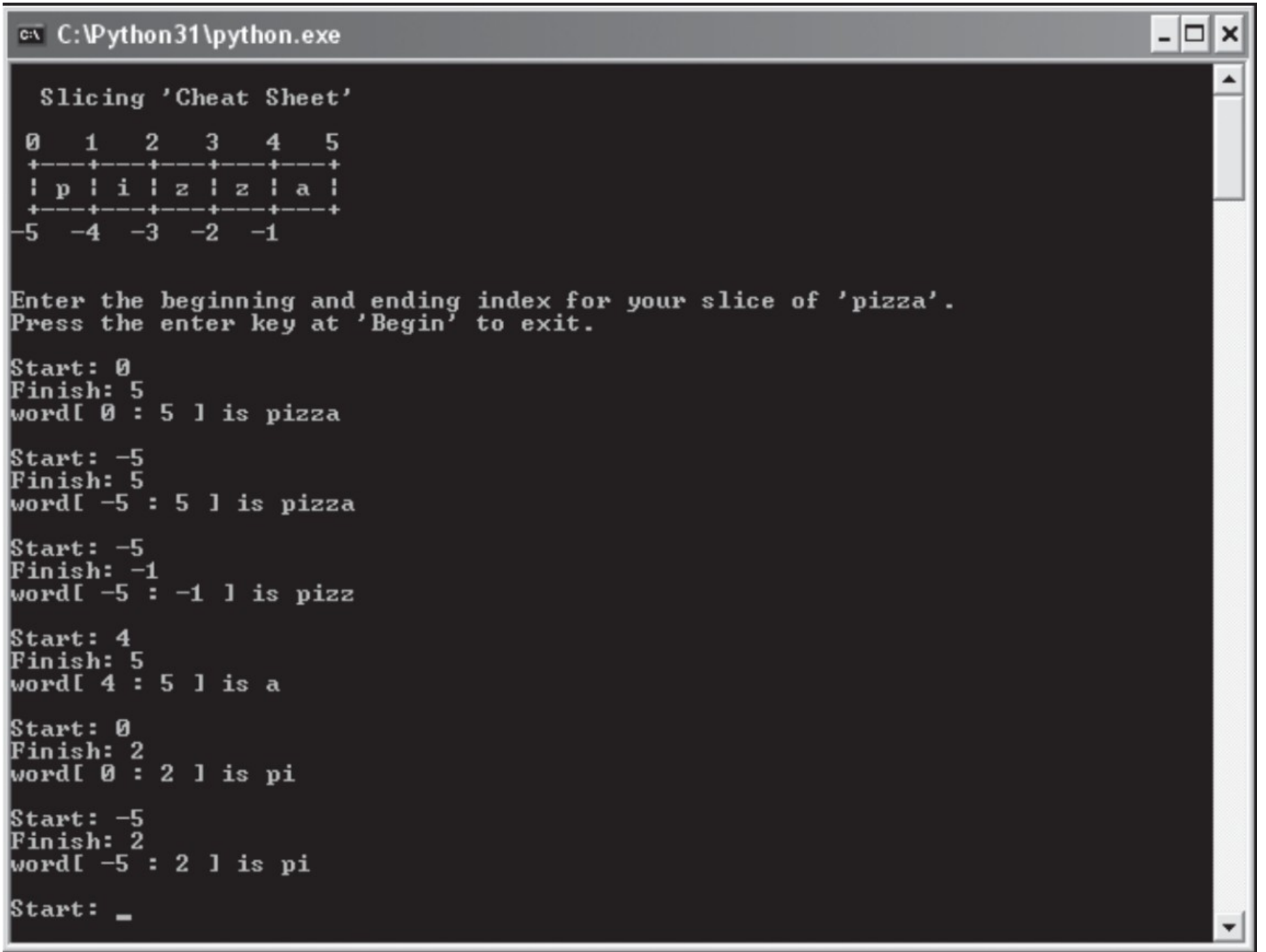
```
for letter in message:
    if letter.lower() not in VOWELS:
        new_message += letter
        print("A new string has been created:",
        new_message)
```

- Python is picky about strings and characters, eg, "A" ≠ "a" .

- To make sure that only lowercase letters is considered, letter**.lower()** is used.

- **new_message += letter** is exactly the same as
  new_message = new_message + letter

Introducing the Pizza Slicer Program



pizza_slicer.py

```
# Pizza Slicer
# Demonstrates string slicing

word = "pizza"

print(
"""
  Slicing 'Cheat Sheet'

 0   1   2   3   4   5
 +---+---+---+---+---+
  | p | i |  z | z | a |
 +---+---+---+---+---+
-5  -4  -3  -2  -1

"""
)
```

```
print("Enter the beginning and ending index for your”,
        “ slice of 'pizza'.")
print("Press the enter key at 'Begin' to exit.")

start = None
start = input("\nStart: ")

while start != "":
    if start:
        start = int(start)

        finish = int(input("Finish: "))

        print("word[", start, ":", finish, "] is", end=" ")
        print(word[start:finish])

    start = (input("\nStart: "))

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

Introducing None

- **None** is Python's way of representing nothing:
- **None** makes a good placeholder for a value.
- **None** also evaluates to False when treated as a condition.
- **None** can be used to initialize a variable for use:

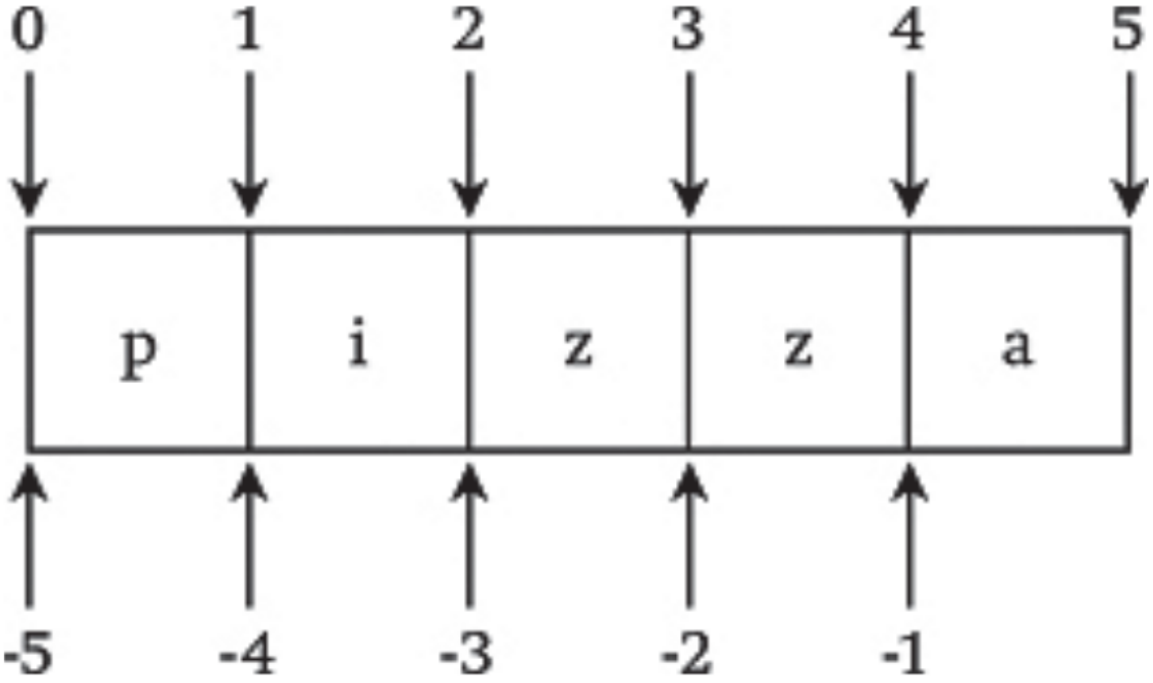
```
start = None
```

Understanding Slicing

- Using indexing, you can copy (or *slice*) one element or part of a sequence.

- To create a slice, you supply a starting position and ending position.

```
>>> word = "pizza"
>>> print(word[0:5])
pizza
>>> print(word[1:3])
iz
>>> print(word[-4:-2])
iz
>>> print(word[-4:3])
iz
```



- If you create an “impossible” slice, like word[2:1] , you won’t cause an error. Instead, Python will quietly return an empty sequence. So be careful!

Using Slicing Shorthand

- You can omit the beginning point for the slice to start the slice at the beginning of the sequence: word[:4] = word[0:4]

- You can omit the ending point so that the slice ends with the very last element: word[2:] = word[2:5]

- You can even omit both numbers to get a slice that is the entire sequence: word[:] = word[0:5] (≠ word)

```
>>> word = "pizza"
>>>print(word[0:4])
pizz
>>>print(word[:4])
pizz
>>>print(word[2:5])
zza
```

```
 >>>print(word[2:])
zza
>>>print(word[0:5])
pizza
>>> print(word[:])
Pizza
```

- it’s that [:] returns a complete copy of a sequence, so this is a quick and efficient way to make a copy.

Creating Tuples

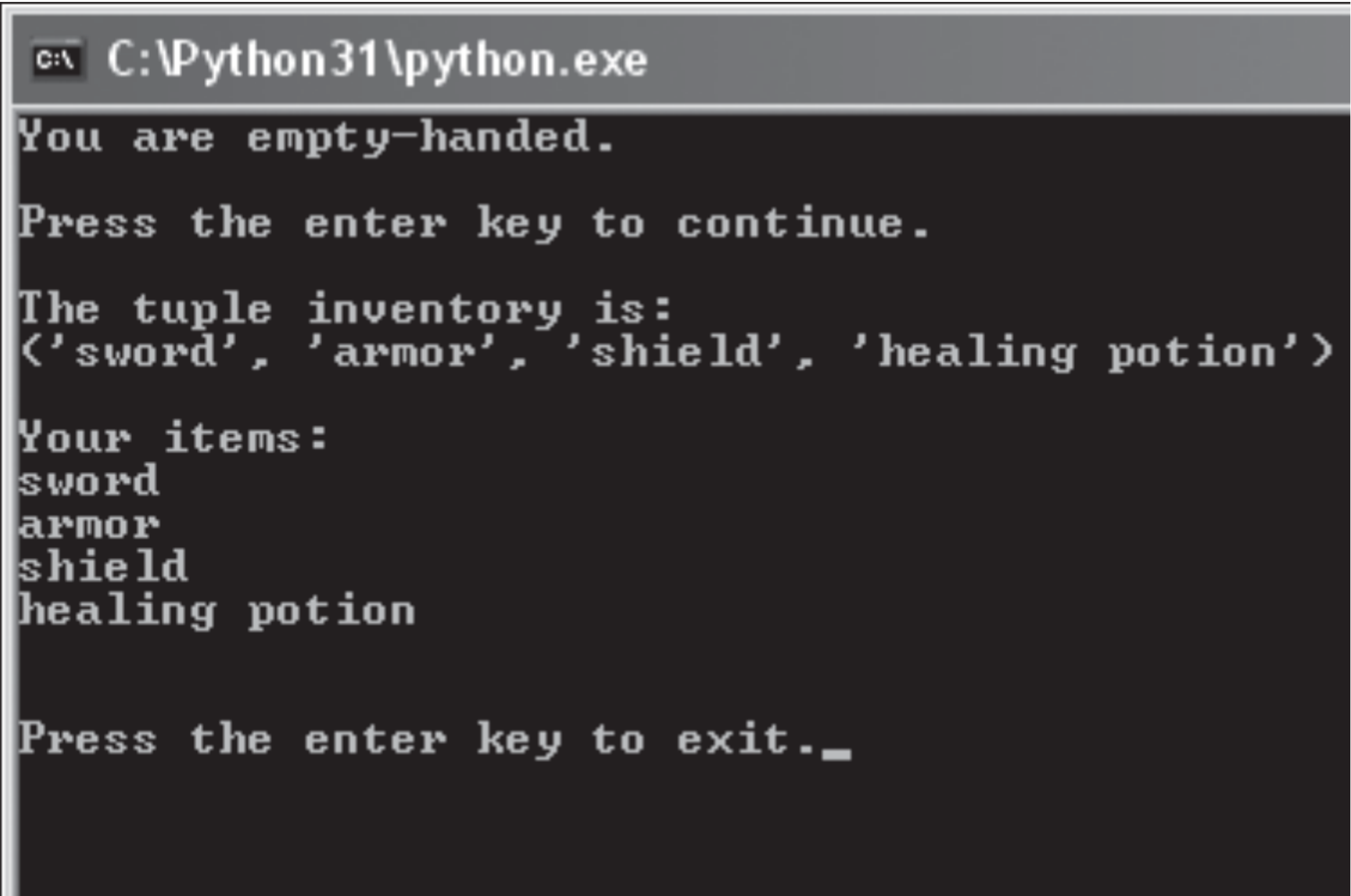
- Tuples are a type of sequence, like strings. But tuples can contain elements of any type.

- Tuple elements don’t have to all be of the same type. You could create a tuple with both strings and numbers.

- You can create a tuple that contains a sequence of graphic images, sound files, or even a group of aliens.

- Whatever you can assign to a variable, you can group together and store as a sequence in a tuple.

Introducing the Hero's Inventory Program



hero's_inventory.py

```
# Hero's Inventory
# Demonstrates tuple creation

# create an empty tuple
inventory = ()

# treat the tuple as a condition
if not inventory:
    print("You are empty-handed.")

input("\nPress the enter key to continue.")

# create a tuple with some items
inventory = ("sword",
             "armor",
             "shield",
             "healing potion")
```

```
# print the tuple
print("\nThe tuple inventory is:")
print(inventory)

# print each element in the tuple
print("\nYour items:")
for item in inventory:
    print(item)

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

Creating an Empty Tuple

- To create a tuple, you just surround a sequence of values, separated by commas, with parentheses.

- Even a pair of lone parentheses is a valid (but empty) tuple:

```
inventory = ()
```

Treating a Tuple as a Condition

- You could treat any value in Python as a condition. That means you can treat a tuple as a condition, too:

```
if not inventory:
    print("You are empty-handed.")
```

- As a condition, an empty tuple is False. A tuple with at least one element is True.

Creating a Tuple with Elements

- Create a new tuple with string elements

```
inventory = ("sword",
             "armor",
             "shield",
             "healing potion")
```

- That makes the 1st element the string "sword" , the next "armor" , the next "shield" , and the last element "healing potion" . So each string is a single element in this tuple.

- Notice that the tuple spans multiple lines. This is one of the few cases where Python lets you break up a statement across multiple lines.

Printing a Tuple

- Though a tuple can contain many elements, you can print the entire tuple just like you would any single value:

```
print("\nThe tuple inventory is:")
print(inventory)
```

Looping Through a Tuple's Elements

- A for loop to march through the elements in inventory and print each one individually:

```
for item in inventory:
    print(item)
```

- Tuples don't have to be filled with values of the same type. A single tuple can just as easily contain strings, integers, and floating-point numbers, for example.

Introducing the Hero's Inventory 2.0

```
C:\Python31\python.exe
Your items:
sword
armor
shield
healing potion

Press the enter key to continue.
You have 4 items in your possession.

Press the enter key to continue.
You will live to fight another day.

Enter the index number for an item in inventory: 1
At index 1 is armor

Enter the index number to begin a slice: 2
Enter the index number to end the slice: 4
inventory[ 2 : 4 ] is ('shield', 'healing potion')

Press the enter key to continue.
You find a chest.  It contains:
('gold', 'gems')
You add the contents of the chest to your inventory.
Your inventory is now:
('sword', 'armor', 'shield', 'healing potion', 'gold', 'gems')


Press the enter key to exit.
```

hero's_inventory2.py

```
# Hero's Inventory 2.0
# Demonstrates tuples

# create a tuple with items and display with a for loop
inventory = ("sword",
             "armor",
             "shield",
             "healing potion")
print("Your items:")
for item in inventory:
    print(item)

input("\nPress the enter key to continue.")

# get the length of a tuple
print("You have", len(inventory),
      "items in your possession.")
input("\nPress the enter key to continue.")
```

```
# test for membership with in
if "healing potion" in inventory:
    print("You will live to fight another day.")

# display one item through an index
index = int(input(\
"\nEnter the index number for an item in inventory: "))
print("At index", index, "is", inventory[index])

# display a slice
start=int(input(\
"\nEnter the index number to begin a slice:"))
finish=int(input(\
"Enter the index number to end the slice: "))

print("inventory[", start, ":", finish, "] is", end=" ")
print(inventory[start:finish])

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

```
# concatenate two tuples
chest = ("gold", "gems")
print("You find a chest.  It contains:")
print(chest)

print("You add the contents of the chest to your”,
      “inventory.")
inventory += chest
print("Your inventory is now:")
print(inventory)

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

Using the len() Function with Tuples

- If you want to know the length of a tuple, place it inside the parentheses of len(). The function returns the number of elements in the tuple.

- Empty tuples, or any empty sequences for that matter, have a length of 0.

```
print("You have", len(inventory),
      "items in your possession.")
```

- Notice that in the tuple inventory , the string "healing potion" is counted as a single element, even though it’s 2 words.

Using the in Operator with Tuples

- You can use the in operator with tuples to test for element membership:

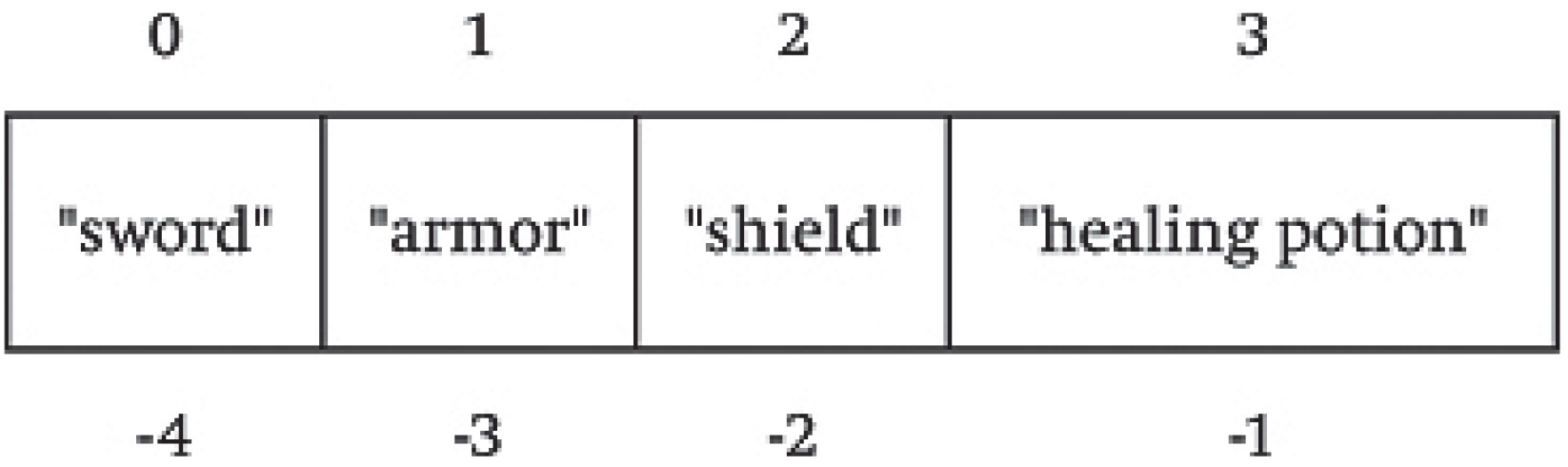
```
if "healing potion" in inventory:
    print("You will live to fight another day.")
```

Indexing Tuples

- Indexing tuples works like indexing strings:

index = int(input("\nEnter the index number for an item in inventory: "))

print("At index", index, "is", inventory[index])

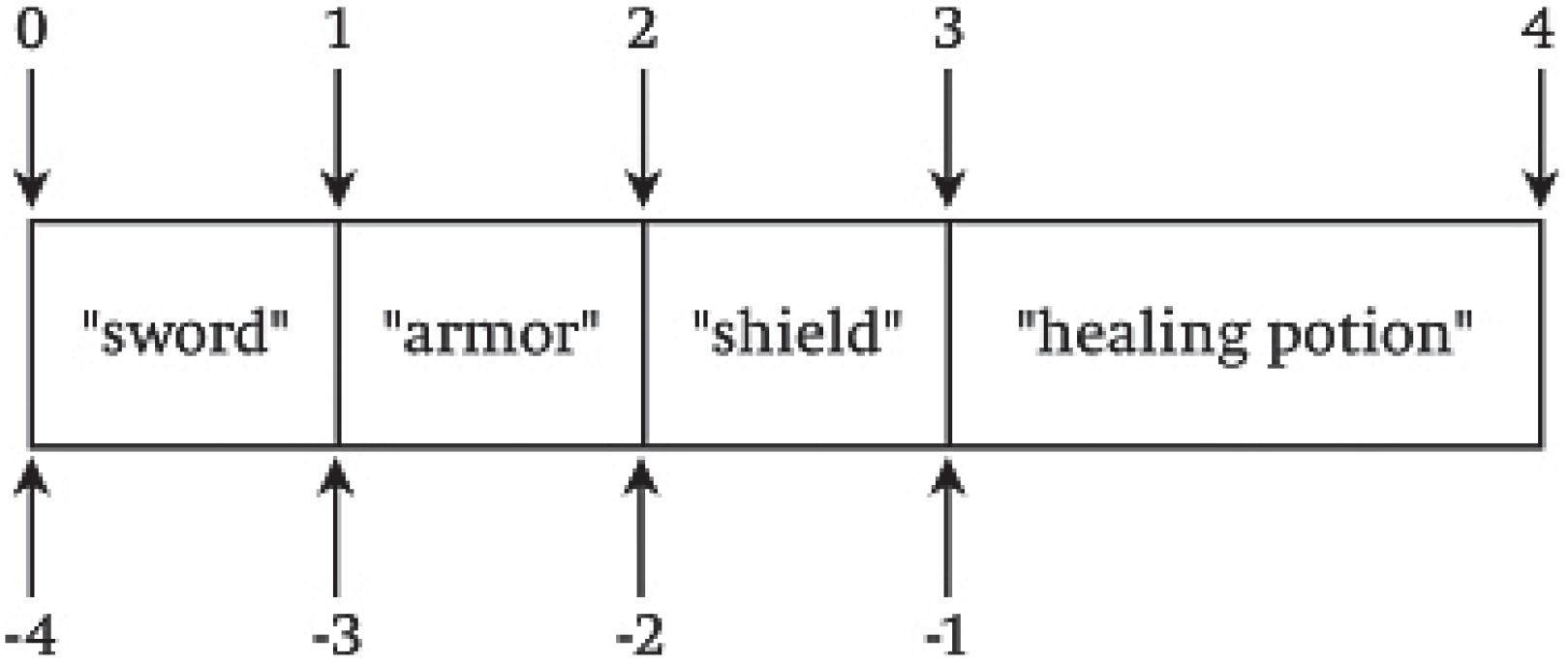


Slicing Tuples

- Slicing works just like you saw with strings:

```
start=int(input(\
        "\nEnter the index number to begin a slice:"))
Finish=int(input(\
        "Enter the index number to end the slice:"))
print("inventory[", start, ":", finish, "] is", end=" ")

print(inventory[start:finish])
```



Understanding Tuple Immutability

- Like strings, tuples are immutable:

```
>>> inventory=("sword", "armor", "shield",
                        "healing potion")

>>> print(inventory)
('sword', 'armor', 'shield', 'healing potion')

>>> inventory[0] = "battleax"
Traceback (most recent call last):
    File "<pyshell#3>", line 1, in ?
      inventory[0] = "battleax"
TypeError: object doesn't support item assignment
```

Concatenating Tuples

- You can concatenate tuples the same way you concatenate strings:

```
chest = ("gold", "gems")
print("You find a chest. It contains:")
print(chest)

print("You add the contents of the chest to your”,
        “ inventory.")
inventory += chest
print("Your inventory is now:")
print(inventory)
```

word_jumble.py

```
# Word Jumble
#
# The computer picks a word and then "jumbles" it
# The player has to guess the original word

import random

# create a sequence of words to choose from
WORDS = ("python", "jumble", "easy", "difficult",
"answer", "xylophone")

# pick one word randomly from the sequence
word = random.choice(WORDS)

# create a variable to see if the guess is correct
correct = word
```

```
# create a jumbled version of the word
jumble =""
while word:
    position = random.randrange(len(word))
    jumble += word[position]
    word = word[:position] + word[(position + 1):]

# start the game
print(
"""
          Welcome to Word Jumble!

   Unscramble the letters to make a word.
(Press the enter key at the prompt to quit.)
"""
)

print("The jumble is:", jumble)
```

```
guess = input("\nYour guess: ")

while guess != correct and guess != "":
    print("Sorry, that's not it.")
    guess = input("Your guess: ")

if guess == correct:
    print("That's it!  You guessed it!\n")

print("Thanks for playing.")

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

Introducing the Word Jumble Game

Setting Up the Program

- Use a tuple to create a sequence of words. Notice that the variable name WORD is in all caps, implying that it will be treated as a constant:

```
WORDS = ("python", "jumble", "easy", "difficult",
         "answer", "xylophone")
```

- Use random.**choice()** to get a random word from WORDS:

```
word = random.choice(WORDS)
```

- random.**choice()** picks a random element from whatever sequence you give.

Planning the Jumble Creation Section

- Algorithm to create a jumbled word from the chosen word:

create an empty jumble word
while the chosen word has letters in it
    extract a random letter from the chosen word
    add the random letter to the jumble word

- Because strings are immutable, one can't actually "extract a random letter" from the string the user entered. But, one can create a new string that doesn't contain the randomly chosen letter.

- Although one can't "add the random letter" to the jumble word string either, but one can create a new string by concatenating the current jumble word with the "extracted" letter.

Setting Up the Loop

- The jumble creation process is controlled by a while loop:

while word:

- The while will continue until word becomes an empty string

- Each time the loop executes, the computer creates a new version of word with one letter “extracted” and assigns it back to word .

- Eventually, word will become the empty string and the jumbling will be done.

Generating a Random Position in word

- The 1st line in the loop body generates a random position in word, based on its length:

position = random.randrange(len(word))

- So, the letter word[position] is the letter that is going to be “extracted” from word and “added to” jumble .

Creating New Versions of jumble & word

- A new version of the string jumble is equal to its old self, plus the letter word[position] :

jumble += word[position]

- Creates a new version of word minus the one letter at position position:

word = word[:position] + word[(position + 1):]

- Using slicing, we creates 2 new strings from word. The 1st one, word[:position] , is every letter up to, but not including, word[position] . The next one, word[(position + 1):] , is every letter after word[position] .

- These 2 strings are joined together and assigned to word , which is now equal to its old self, minus the one letter word[position] .

Getting the Player's Guess

- The computer keeps asking the player for a guess as long as the player doesn't enter the correct word or press the Enter key at the prompt:

```
guess = input("\nYour guess: ")

while guess != correct and guess != "":
        print("Sorry, that's not it.")
        guess = input("Your guess: ")
```

Congratulating the Player

- If the player has guessed the word, then the computer offers its hearty congratulations:

```
if guess == correct:
    print("That's it!  You guessed it!\n")
```

Quiz 4: Create a game where the computer picks a random word and the player has to guess that word. The computer tells the player how many letters are in the word. Then the player gets 5 chances to ask if a letter is in the word. The computer can only respond with “yes” or “no”. Then, the player must guess the word.