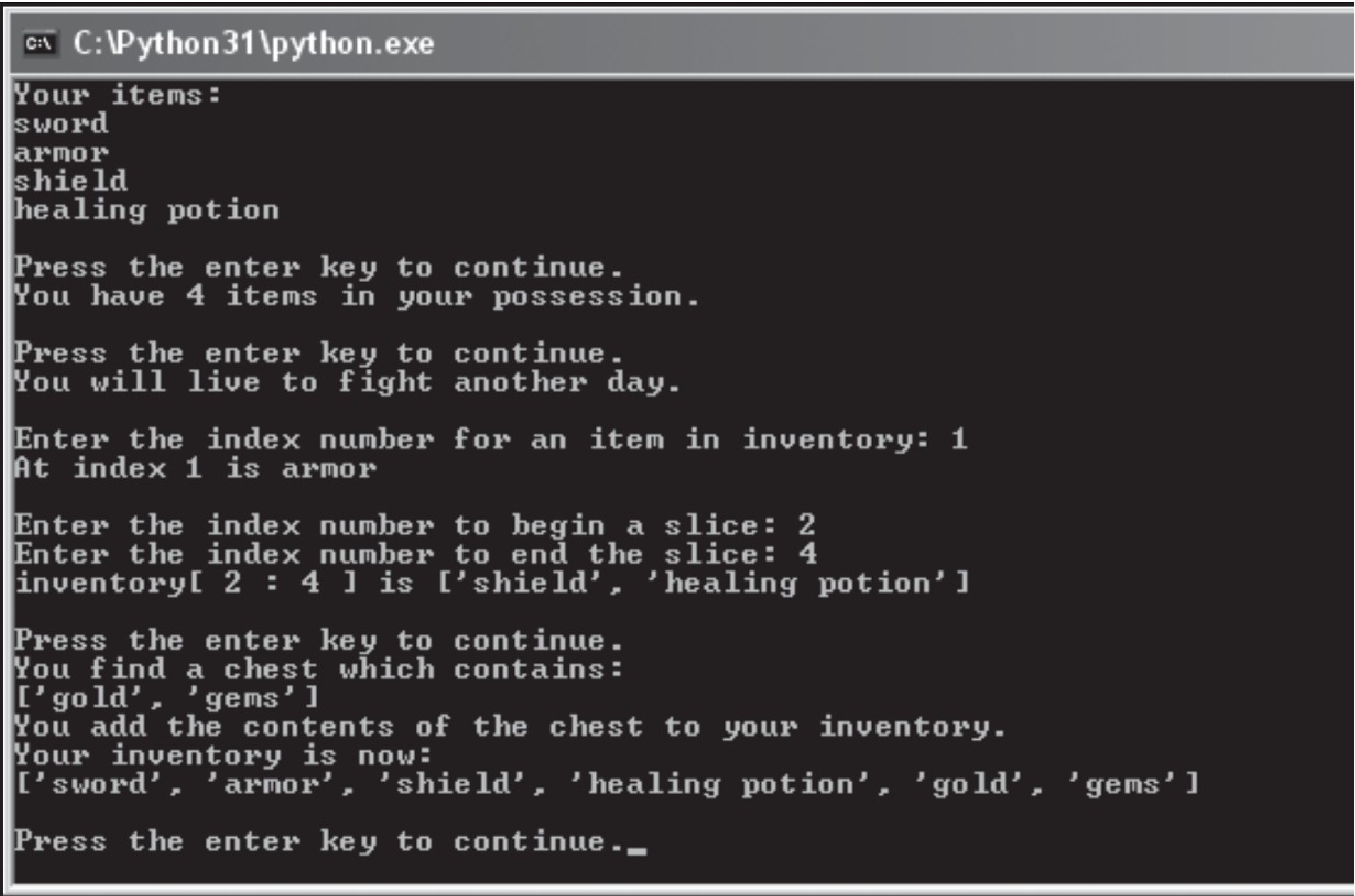
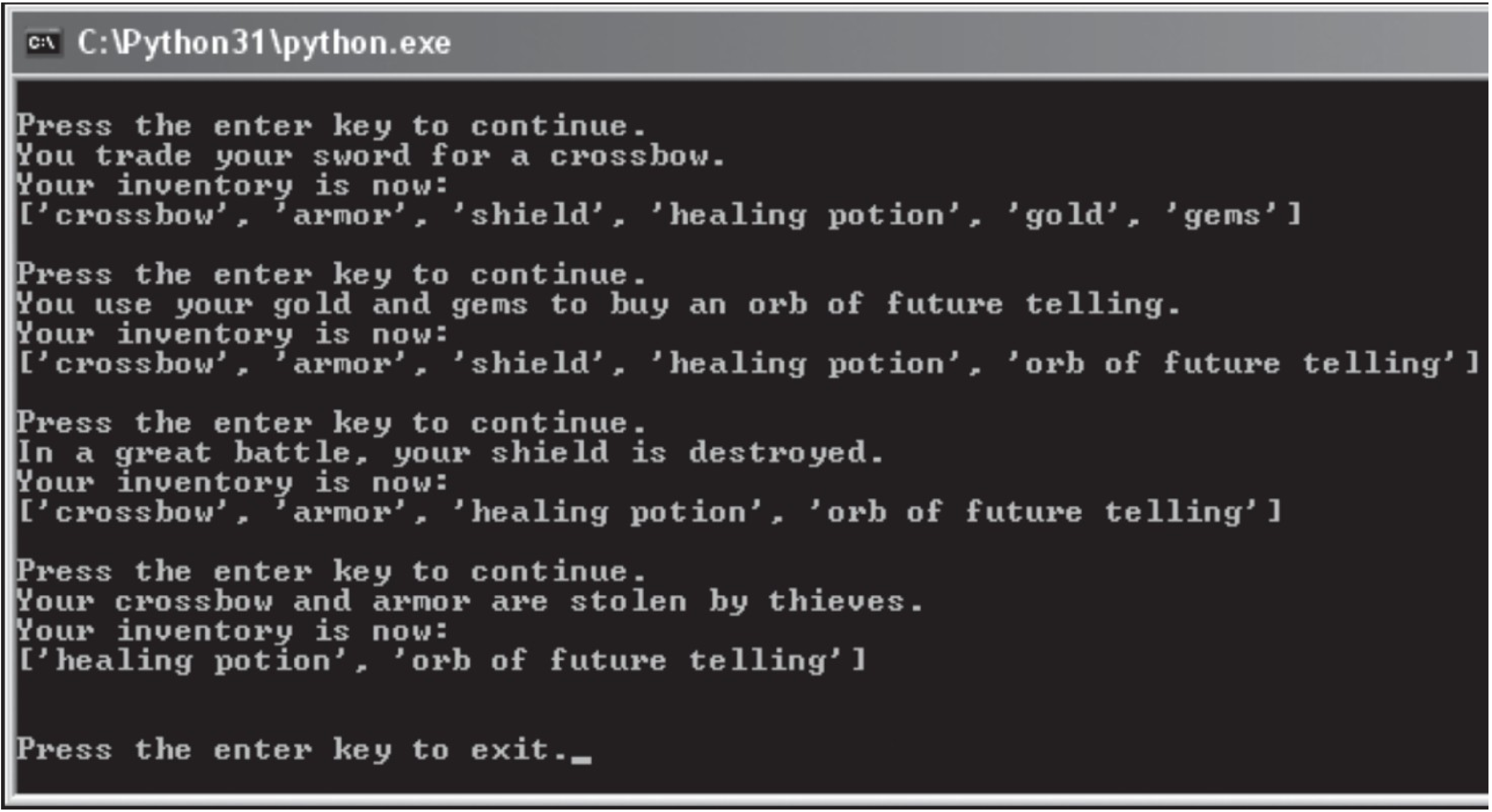
Chapter 5

Lists and Dictionaries: The Hangman Game

The Hero’s Inventory 3.0 Program



```
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 which 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 continue._
```



```
Press the enter key to continue.
You trade your sword for a crossbow.
Your inventory is now:
['crossbow', 'armor', 'shield', 'healing potion', 'gold', 'gems']

Press the enter key to continue.
You use your gold and gems to buy an orb of future telling.
Your inventory is now:
['crossbow', 'armor', 'shield', 'healing potion', 'orb of future telling']

Press the enter key to continue.
In a great battle, your shield is destroyed.
Your inventory is now:
['crossbow', 'armor', 'healing potion', 'orb of future telling']

Press the enter key to continue.
Your crossbow and armor are stolen by thieves.
Your inventory is now:
['healing potion', 'orb of future telling']


Press the enter key to exit._
```

hero's_inventory3.py

```
# Hero's Inventory 3.0
# Demonstrates lists
# create a list 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 list
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 lists
chest = ["gold", "gems"]
print("You find a chest which contains:")
print(chest)
print("You add the contents of the chest to your
inventory.")
inventory += chest
```

```
print("Your inventory is now:")
print(inventory)

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

# assign by index
print("You trade your sword for a crossbow.")
inventory[0] = "crossbow"
print("Your inventory is now:")
print(inventory)

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

# assign by slice
print("You use your gold and gems to buy an orb “,
        “of future telling.")
inventory[4:6] = ["orb of future telling"]
print("Your inventory is now:")
print(inventory)

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

```
# delete an element
print("In a great battle, your shield is destroyed.")
del inventory[2]
print("Your inventory is now:")
print(inventory)

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

# delete a slice
print("Your crossbow and armor are stolen by thieves.")
del inventory[:2]
print("Your inventory is now:")
print(inventory)

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

Creating a List

- Lists are sequences, just like tuples—but lists are mutable. They can be modified. So, lists can do everything tuples can, plus more.

- To create a new list, assign it to inventory , and print each element:

```
inventory = ["sword","armor","shield","healing potion"]
```

- To create a list instead of a tuple, surrounded the elements with **square brackets** instead of parentheses.

Using the len() Function and the in Operator with Lists

- The **len()** function works the same with lists as it does with tuples:

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

- The **in** operator works the same with lists as it does with tuples:

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

Indexing Lists and Slicing Lists

- Indexing a list is the same as indexing a tuple—just supply the position number of the element you’re after in brackets:

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

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

- slicing a list is exactly the same as slicing a tuple:

```
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])
```

Concatenating Lists

- Concatenating lists works the same way concatenating tuples does.

- The only real difference here is that you can only concatenate sequences of the same type:

```
chest = ["gold", "gems"]

print("You find a chest which contains:")
print(chest)

print("You add the contents of the chest to your “,
        “inventory.")

inventory += chest

print("Your inventory is now:")
print(inventory)
```

Assigning a New List Element by Index

- There is one huge difference between List and Tuple. Lists are mutable. They can change.

- Because lists are mutable, you can assign an existing element a new value:

```
print("You trade your sword for a crossbow.")
inventory[0] = "crossbow"
print("Your inventory is now:")
print(inventory)
```

- The code assigns the string "crossbow" to the element in inventory at position 0.

- The new string replaces the previous value (ie, "sword").

- You can assign an existing list element a new value with indexing, but you can’t create a new element in this way.

Assigning a New List Slice

- You can assign a new value to a slice.

- Assign the list ["orb of future telling"] to the slice inventory[4:6] :

inventory[4:6] = ["orb of future telling"]

- The statement replaces the 2 items inventory[4] and inventory[5] with the string "orb of future telling" .

- Because we assigned a list with one element to a slice with 2 elements, the length of the list shrunk by 1.

Deleting a List Element

- You can delete an element from a list with **del**:

```
del inventory[2]
```

- The element that was at position number 2, the string "shield" , is then removed from inventory .

- The length of the list shrinks by 1, and all of the elements after the deleted one “slide down” one position.

- So there is still an element in position 2; it’s just the element that was at position 3.

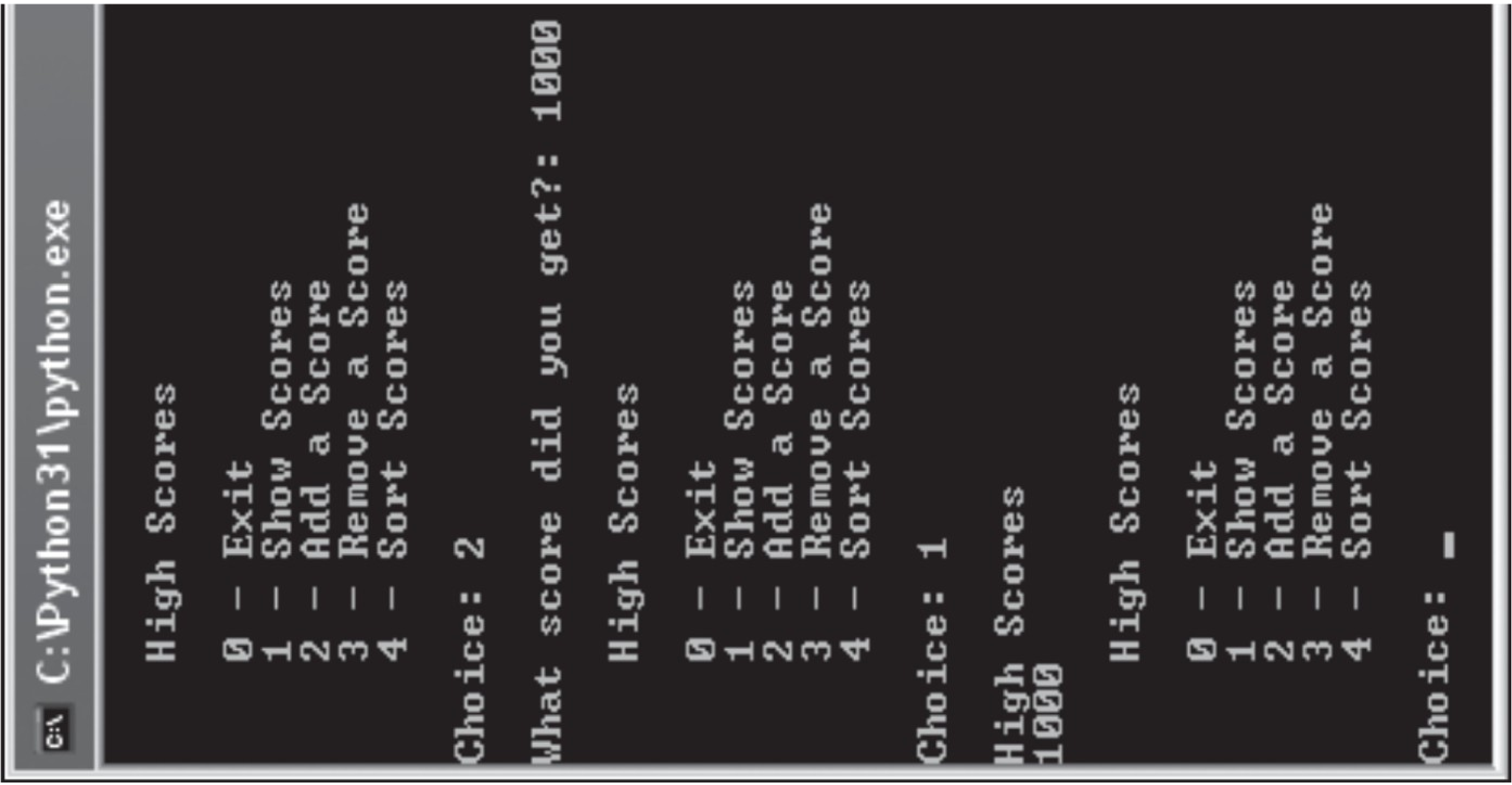
Deleting a List Slice

- The following line removes the slice inventory[:2] , which is ["crossbow", "armor"] , from inventory :

```
del inventory[:2]
```

- Just as with deleting an element, the length of the list shrinks and the remaining elements form a new, continuous list, starting from position 0.

Introducing the High Scores Program



```
C:\Python31\python.exe

    High Scores

      0 - Exit
      1 - Show Scores
      2 - Add a Score
      3 - Remove a Score
      4 - Sort Scores

Choice: 2
What score did you get?: 1000

    High Scores

      0 - Exit
      1 - Show Scores
      2 - Add a Score
      3 - Remove a Score
      4 - Sort Scores

Choice: 1
High Scores
1000

    High Scores

      0 - Exit
      1 - Show Scores
      2 - Add a Score
      3 - Remove a Score
      4 - Sort Scores

Choice: _
```

high_scores.py

```
# High Scores
# Demonstrates list methods

scores = []
choice = None

while choice != "0":
    print(
    """
    High Scores

    0 - Exit
    1 - Show Scores
    2 - Add a Score
    3 - Remove a Score
    4 - Sort Scores
    """
    )
```

```
choice = input("Choice: ")
print()

# exit
if choice == "0":
    print("Good-bye.")

# list high-score table
elif choice == "1":
    print("High Scores")
    for score in scores:
        print(score)

# add a score
elif choice == "2":
    score = int(input("What score did you get?: "))
    scores.append(score)
    # remove a score
    elif choice == "3":
        score = int(input("Remove which score?: "))
        if score in scores:
            scores.remove(score)
        else:
            print(score, "isn't in the high scores list.")

    # sort scores
    elif choice == "4":
        scores.sort(reverse=True)

    # some unknown choice
    else:
        print("Sorry, but", choice, "isn't a valid choice.")

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

Setting Up the Program

- We create 2 variables. scores is a list that will contain the scores. I set it to an empty list to start out. choice represents the user's choice from the menu. I initialized it to None :

```
scores = []
choice = None
```

Displaying the Menu

- The while loop is the core of the program. It continues until the user enters 0. The rest of this code prints the menu and gets the user’s choice:

```
while choice != "0":
    print(
    """
    High Scores

    0 - Exit
    1 - Show Scores
    2 - Add a Score
    3 - Remove a Score
    4 - Sort Scores
    """
    )
    choice = input("Choice: ")
    …
```

Exiting the Program

- Check first if the user wants to quit. If the user enters 0, the computer says “Good-bye.”:

```
if choice == "0":
    print("Good-bye.")
```

- If the user enters 0, then the while loop’s condition will be false the next time it’s tested. The loop will end and so will the program.

Displaying the Scores and Adding a Score

- If the user enters 1, then this elif block executes and the computer displays the scores:

```
elif choice == "1":
    print("High Scores")
    for score in scores:
      print(score)
```

- If the user enters 2, the user is then asked for a new score and assigns it to score .

- The last line uses the **append()** list method to add score to the end of scores. The list becomes one element longer:

```
elif choice == "2":
    score = int(input("What score did you get?: "))
    scores.append(score)
```

Removing a Score

- When the user enters 3, the computer gets a score from the user to remove.

- If the score is in the list, the 1st occurrence of it is removed. If the score isn't in the list, the user is informed:

```
elif choice == "3":
    score = int(input("Remove which score?: "))
    if score in scores:
        scores.remove(score)
    else:
        print(score, "isn't in the high scores list.")
```

- The code first checks if the score is in the list. If it is, the list method **remove()** is invoked.

- The method goes through the list, starting at position 0, and searches for the value passed to score.

- When the method finds the 1st occurrence of the value, that element is deleted from the list.

- If the value is in the list more than once, only the 1st one is removed.

- If the method successfully removes an element from the list, the list becomes one element shorter.

- **remove()** is different from del. The **remove()** method doesn't delete an element based on a position, but rather on a value.

- If you try to remove a value that isn't in a list with the **remove()** method, you'll generate an error. A safer way to do it is:

```
if score in scores:
    scores.remove(score)
```

Sorting the Scores

- The **sort()** method sorts the elements in the list:

```
elif choice == "4":
    scores.sort(reverse=True)
```

- By default **sort()** orders the elements in ascending—smallest values first.

- You can tell the **sort()** method to sort values in descending order—largest values first by passing True to the method’s **reverse** parameter.

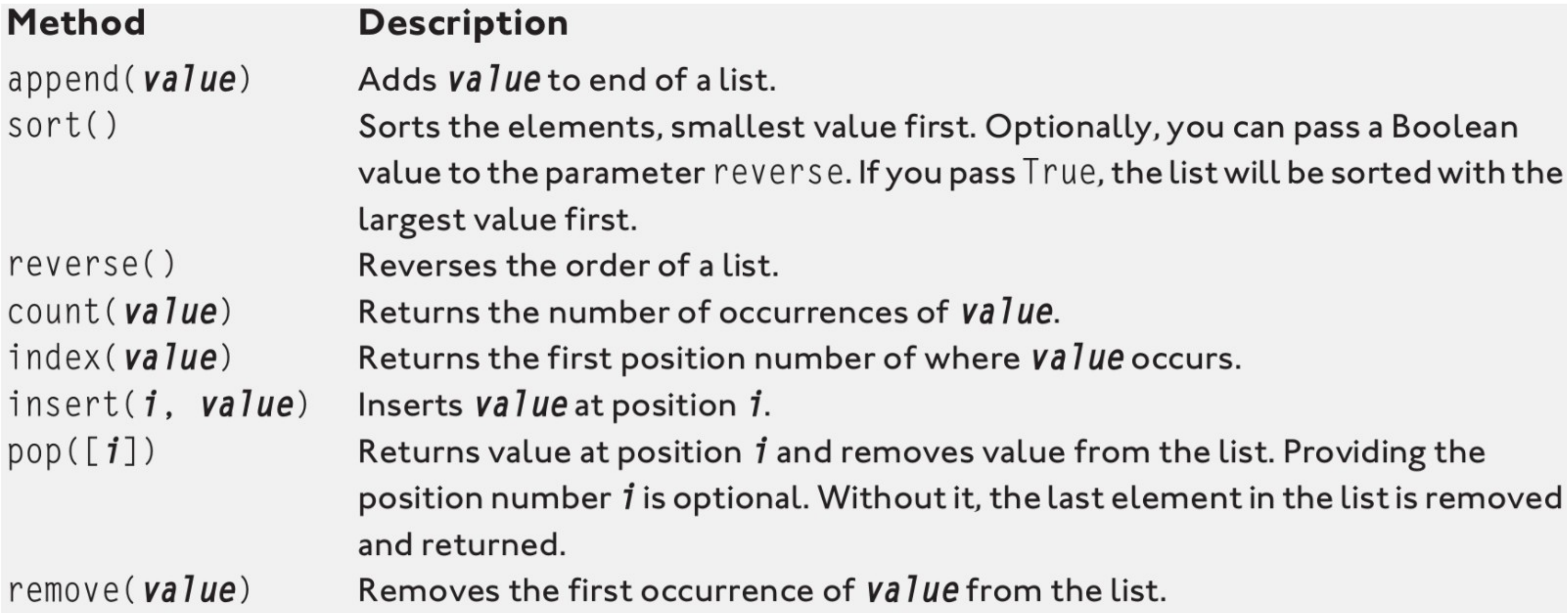
- If you want to sort a list in ascending order, you can simply call the method without passing values

```
numbers.sort()
```

Dealing with an Invalid Choice

- If the user enters a number that isn’t a valid choice, the else clause catches it:

```
    else:
        print("Sorry, but", choice, "isn't a valid choice.")
```



Method	Description
append(*value*)	Adds *value* to end of a list.
sort()	Sorts the elements, smallest value first. Optionally, you can pass a Boolean value to the parameter reverse. If you pass True, the list will be sorted with the largest value first.
reverse()	Reverses the order of a list.
count(*value*)	Returns the number of occurrences of *value*.
index(*value*)	Returns the first position number of where *value* occurs.
insert(*i*, *value*)	Inserts *value* at position *i*.
pop([*i*])	Returns value at position *i* and removes value from the list. Providing the position number *i* is optional. Without it, the last element in the list is removed and returned.
remove(*value*)	Removes the first occurrence of *value* from the list.

Understanding When to Use Tuples Instead of Lists

- There are a few occasions where tuples make more sense than lists:

  * Tuples are faster than lists because the they won’t change.

  * Tuples’ immutability makes them perfect for creating constants since they can’t change.

  * Sometimes tuples are required, eg, Python requires immutable values.

- But, because lists are so flexible, you’re probably best off using them rather than tuples the majority of the time.

Creating Nested Sequences

- Lists can contain other lists/tuples, and tuples can contain other tuples/lists. They're called *nested sequences*.

- With nested sequences, you include entire lists or tuples as elements:

```
>>> nested=["1st", ("2nd","3rd"), ["4th",“5fth","6th"]]
>>> print(nested)
['1st', ('2nd', '3rd'), ['4th', '5th', '6th']]
```

- Although we see 6 strings here, nested has only 3 elements. The 1st element is the string "1st" , the 2nd element is the tuple ("2nd", "3rd") , and the 3rd element is the list ["4th", "5th", "6th"] .

- While you can create a list or tuple with any number of lists and tuples, useful nested sequences often have a consistent pattern:

```
>>> scores = [("Moe", 1000), ("Larry", 1500),
                    ("Curly",3000)]
>>> print(scores)
[('Moe',1000), ('Larry',1500), ('Curly',3000)]
```

- scores is a list with 3 elements. Each element is a tuple. Each tuple has exactly 2 elements, a string and a number.

- Although you can create nested sequences inside nested sequences many times over, this usually isn’t a good idea:

```
nested = ("deep",("deeper",("deepest","still deepest")))
```

Accessing Nested Elements

- Access elements of a nested sequence through indexing:

```
>>> scores = [("Moe",1000), ("Larry",1500),
                    ("Curly",3000)]
>>> print(scores[0])
('Moe', 1000)
>>> print(scores[1])
('Larry', 1500)
>>> print(scores[2])
('Curly', 3000)
```

- Each element is a tuple. To access one of the elements of one of the tuples, one way is to assign the tuple to a variable and index it:

```
>>> a_score = scores[2]
>>> print(a_score)
('Curly ', 3000)
>>> print(a_score[0])
Curly
```

- There’s a direct way to access "Curly" right from scores :

```
>>> print(scores[2][0])
Curly
```

- By supplying 2 indices with scores[2][0], the computer gets the element from scores at position 2 (ie, ("Curly", 3000)) and, from that, to get the element at position 0 (ie, "Curly").

- We can use this kind of multiple indexing with nested sequences to get directly to a nested element.

Unpacking a Sequence

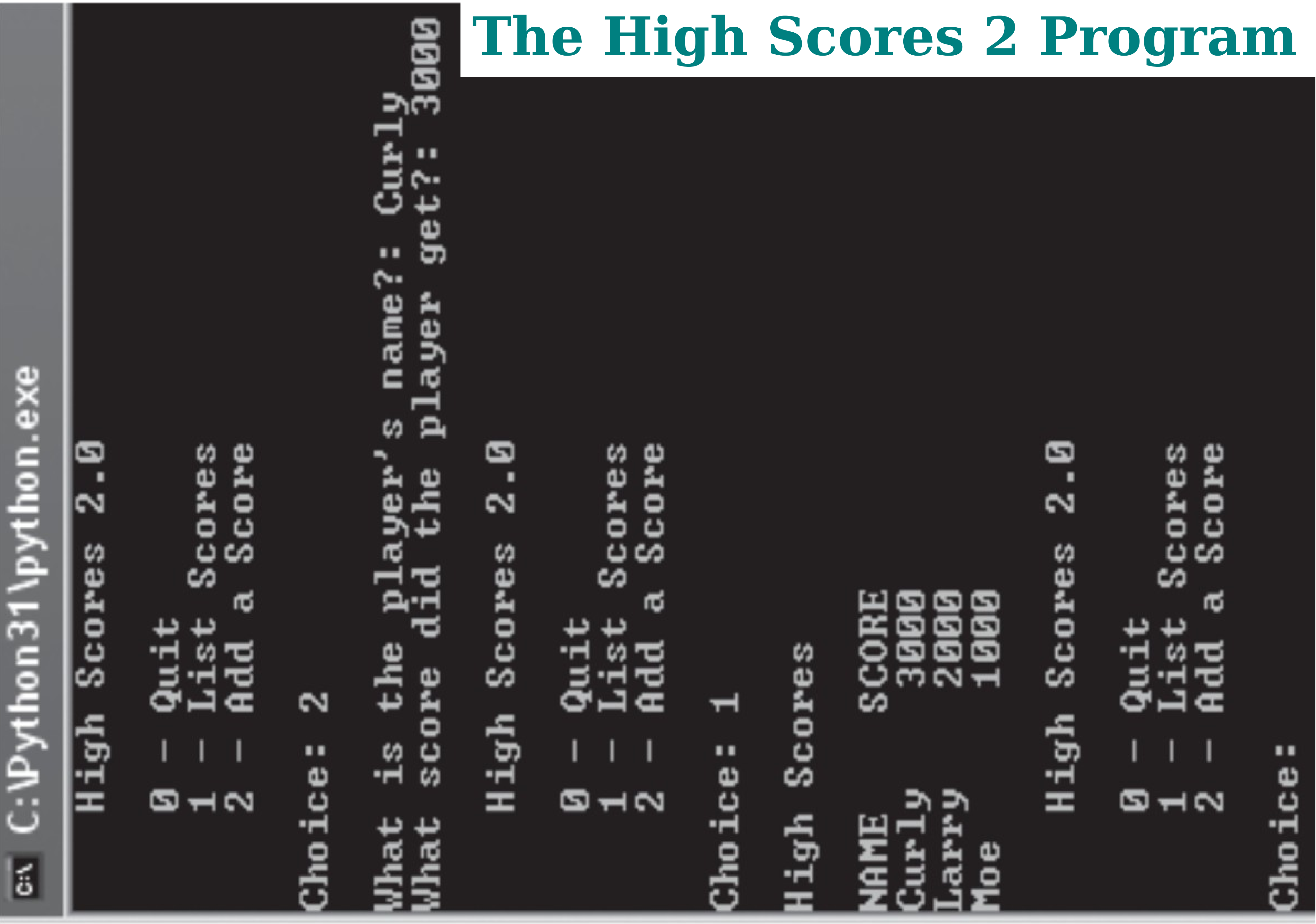
- If we know how many elements are in a sequence, we can assign each to its own variable in a single line of code:

```
>>> name, score = ("Shemp", 175)
>>> print(name)
Shemp
>>> print(score)
175
```

- This is called *unpacking* and works with **any** sequence type.

- Use the same number of variables as elements in the sequence, or you’ll generate an error.

The High Scores 2 Program



```
    High Scores 2.0

    0 - Quit
    1 - List Scores
    2 - Add a Score

Choice: 2

What is the player's name?: Curly
What score did the player get?: 3000

    High Scores 2.0

    0 - Quit
    1 - List Scores
    2 - Add a Score

Choice: 1

High Scores

NAME    SCORE
Curly    3000
Larry    2000
Moe      1000

    High Scores 2.0

    0 - Quit
    1 - List Scores
    2 - Add a Score

Choice:
```

high_scores2.py

```
# High Scores 2.0
# Demonstrates nested sequences

scores = []
choice = None

while choice != "0":

    print(
    """
    High Scores 2.0

    0 - Quit
    1 - List Scores
    2 - Add a Score
    """
    )
```

```
choice = input("Choice: ")
print()

# exit
if choice == "0":
    print("Good-bye.")

# display high-score table
elif choice == "1":
    print("High Scores\n")
    print("NAME\tSCORE")
    for entry in scores:
        score, name = entry
        print(name, "\t", score)
    # add a score
    elif choice == "2":
        name=input("What is the player's name?: ")
        score=int(input("What score did the player get?:"))
        entry = (score, name)
        scores.append(entry)
        scores.sort(reverse=True)
        scores = scores[:5]            # keep only top 5 scores

    # some unknown choice
    else:
        print("Sorry, but", choice, "isn't a valid choice.")

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

Displaying the Scores by Accessing Nested Tuples

- If the user enters 1, the computer goes through each element in scores and unpacks the score and name into the variables score and name:

```
elif choice == "1":
    print("High Scores\n")
    print("NAME\tSCORE")
    for entry in scores:
        score, name = entry
        print(name, "\t", score)
```

Adding a Score by Appending a Nested Tuple

- If the user enters 2, the computer lets the user enter a new score and name. With these 2 values, the computer creates a tuple, entry.

- Then appends this new entry to the list. The computer sorts the list and reverses it so that the highest scores are first.

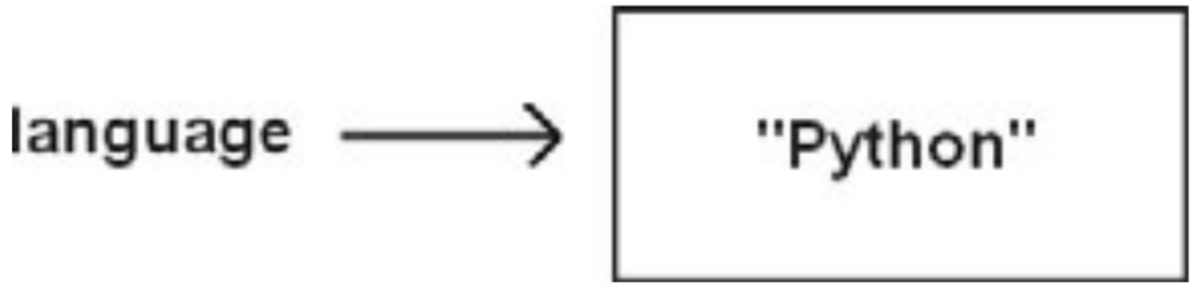
- The final statement slices and assigns the list so that only the top 5 scores are kept.

```
elif choice == "2":
    name=input("What is the player's name?: ")
    score=int(input("What is the player’s score?:"))
    entry = (score, name)
    scores.append(entry)
    scores.sort(reverse=True)
    scores = scores[:5]          # keep only top 5 scores
```

Understanding Shared References

- A variable doesn’t store a copy of a value, but just refers to the place in your computer’s memory where the value is stored.

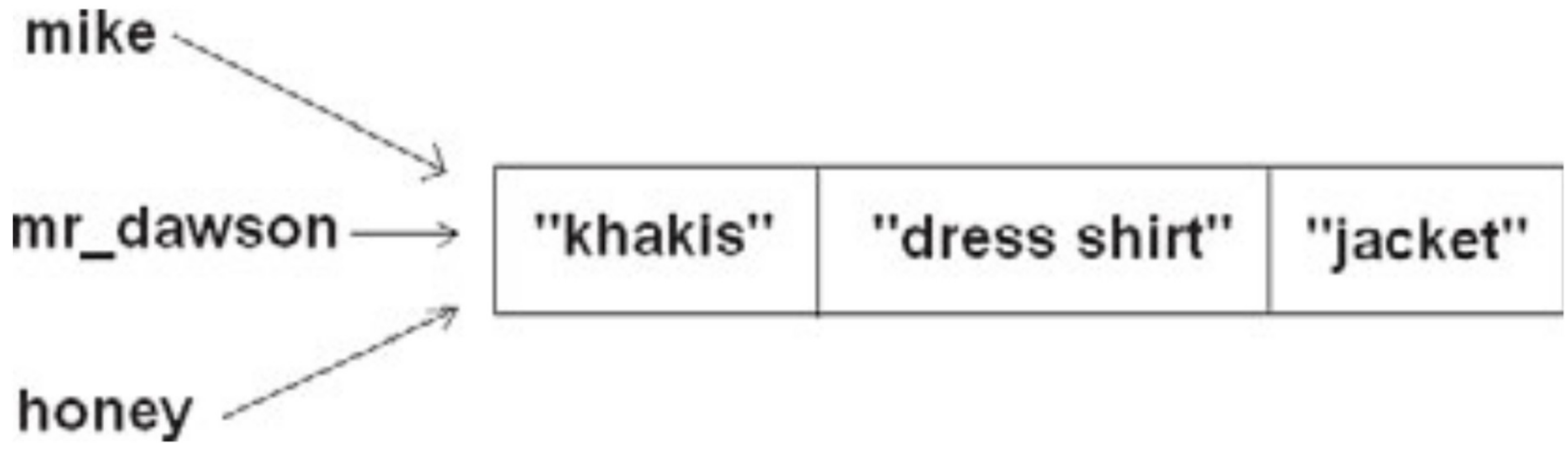
- language = "Python" stores the string "Python" in your computer’s memory and then creates the variable language , which refers to that place in memory.



- When several variables refer to the same mutable value, they share the same reference. They all refer to the one, single copy of that value. And a change to the value through one of the variables results in a change for all the variables, since there is only one, shared copy to begin with.

- Here’s an example to show that 3 variables could all refer to the same list:

```
>>> mike = ["khakis", "dress shirt", "jacket"]
>>> mr_dawson = mike
>>> honey = mike
>>> print(mike)
['khakis', 'dress shirt', 'jacket']
>>> print(mr_dawson)
['khakis', 'dress shirt', 'jacket']
>>> print(honey)
['khakis', 'dress shirt', 'jacket']
```



- This means that a change to the list using any of these 3 variables will change the list they all refer to:

```
>>> honey[2] = "red sweater"
>>> print(honey)
['khakis', 'dress shirt', 'red sweater']
>>> print(mike)
['khakis', 'dress shirt', 'red sweater']
>>> print(mr_dawson)
['khakis', 'dress shirt', 'red sweater']
```

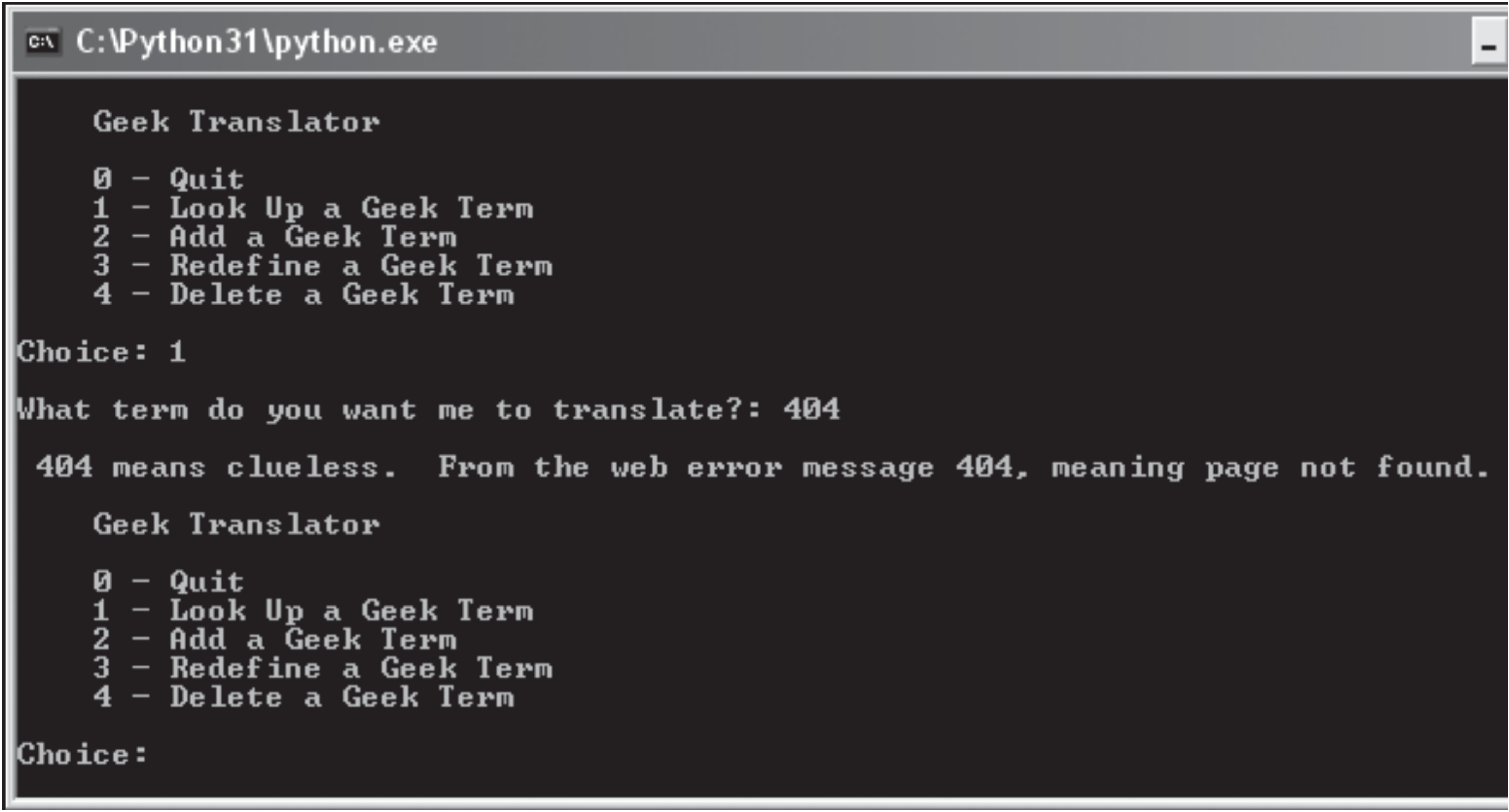
- So be aware of shared references when using mutable values. If you change the value through one variable, it will be changed for all.

- You can avoid this effect if you make a copy of a list through slicing:

```
>>> mike = ["khakis", "dress shirt", "jacket"]
>>> honey = mike[:]
>>> honey[2] = "red sweater"
>>> print(honey)
['khakis', 'dress shirt', 'red sweater']
>>> print(mike)
['khakis', 'dress shirt', 'jacket']
```

- Here, honey is assigned a copy of mike . honey does not refer to the same list. Instead, it refers to a copy. So, a change to honey has no effect on mike .

Introducing the Geek Translator Program



C:\Python31\python.exe

```
    Geek Translator

    0 - Quit
    1 - Look Up a Geek Term
    2 - Add a Geek Term
    3 - Redefine a Geek Term
    4 - Delete a Geek Term

Choice: 1

What term do you want me to translate?: 404

 404 means clueless.  From the web error message 404, meaning page not found.

    Geek Translator

    0 - Quit
    1 - Look Up a Geek Term
    2 - Add a Geek Term
    3 - Redefine a Geek Term
    4 - Delete a Geek Term

Choice:
```

geek_translator.py

```
# Geek Translator
# Demonstrates using dictionaries

geek = {"404": "clueless.  From the web error message \
404, meaning page not found.",
              "Googling": "searching the Internet for \
background information on a person.",
              "Keyboard Plaque" : "the collection of debris \
found in computer keyboards.",
              "Link Rot" : "the process by which web page \
links become obsolete.",
              "Percussive Maintenance" : "the act of striking \
an electronic device to make it work.",
              "Uninstalled" : "being fired.  Especially popular \
during the dot-bomb era."}

choice = None
```

```
while choice != "0":

    print(
    """
    Geek Translator

    0 - Quit
    1 - Look Up a Geek Term
    2 - Add a Geek Term
    3 - Redefine a Geek Term
    4 - Delete a Geek Term
    """
    )

    choice = input("Choice: ")
    print()

    # exit
    if choice == "0":
        print("Good-bye.")
```

```
 # get a definition
elif choice == "1":
    term = input(\
    "What term do you want me to translate?: ")
    if term in geek:
        definition = geek[term]
        print("\n", term, "means", definition)
    else:
        print("\nSorry, I don't know", term)

# add a term-definition pair
elif choice == "2":
    term = input("What term do you want me to add?: ")
    if term not in geek:
        definition = input("\nWhat's the definition?: ")
        geek[term] = definition
        print("\n", term, "has been added.")
    else:
        print("\nThat term already exists! “,
                 ”Try redefining it.")
```

```
# redefine an existing term
elif choice == "3":
    term = input(\
    "What term do you want me to redefine?: ")
    if term in geek:
        definition = input("What's the new definition?: ")
        geek[term] = definition
        print("\n", term, "has been redefined.")
    else:
        print("\nThat term doesn't exist!  Try adding it.")

# delete a term-definition pair
elif choice == "4":
    term=input("What term do you want me to delete?")
    if term in geek:
        del geek[term]
        print("\nOkay, I deleted", term)
    else:
        print("\nI can't do that!", term,
                 "doesn't exist in the dictionary.")
    # some unknown choice
    else:
        print("\nSorry, but", choice, "isn't a valid choice.")

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

Creating Dictionaries

- This following code creates a dictionary named geek. It consists of 6 pairs, called *items*:

```
geek = {"404": "clueless.  From the web error message\
404, meaning page not found.",
              "Googling": "searching the Internet for\
background information on a person.",
              "Keyboard Plaque" : "the collection of debris\
found in computer keyboards.",
              "Link Rot" : "the process by which web page\
links become obsolete.",
              "Percussive Maintenance" : "the act of
\striking an electronic device to make it work.",
              "Uninstalled" : "being fired.  Especially\
popular during the dot-bomb era."}
```

- Each item is made up of a key and a value. The keys are on the left side of the colons. The values are on the right.

Using a Key to Retrieve a Value

- The simplest way to retrieve a value from a dictionary is by directly accessing it with a key.

- To get a key’s value, just put the key in brackets, following the name of the dictionary:

```
>>> geek["404"]
'clueless. From the web error message 404, meaning
page not found. '
>>> geek["Link Rot"]
'the process by which web page links become obsolete. '
```

- When you index a sequence, you use a position number. When you look up a value in a dictionary, you use a key. Dictionaries don’t have position numbers at all.

- **A value can’t be used to get a key in a dictionary.**

Testing for a Key with the in Operator Before Retrieving a Value

- Since using a nonexistent key can lead to an error, one thing you can do is check to see if a key exists before attempting to retrieve its value:

```
>>> if "Dancing Baloney" in geek:
          print("I know what Dancing Baloney is. ")
    else:
          print("I have no idea what Dancing Baloney is.")

I have no idea what Dancing Baloney is.
```

- Using the **in** operator with dictionaries is much the same way you've used it with lists and tuples.

- **in** only checks for keys; it can't check for values used this way.

Using the get() Method to Retrieve a Value

- The dictionary method **get()** has a built-in safety net for handling situations for a value of a key that doesn't exist.

- If the key doesn't exist, **get()** returns a default value, which you can define:

```
>>> print(geek.get("Dancing Baloney",
                   "I have no idea."))
I have no idea.
```

- To use the **get()** method, all you have to do is supply the key you're looking for followed by an optional default value. If the key is in the dictionary, you get its value. If the key isn't in the dictionary, you get the default value.

- If you don't supply a default value, then you get back None:

```
>>> print(geek.get("Dancing Baloney"))
None
```

Getting a Value

If the user enters 1, the code asks for a term to look up. The computer checks to see if the term is in the dictionary. If yes, the program accesses the dictionary, using the term as the key, gets its definition, and prints it out. If the term is not in the dictionary, the computer informs the user:

```
elif choice == "1":
    term=input("What term do you want to translate?")
    if term in geek:
        definition = geek[term]
        print("\n", term, "means", definition)
    else:
        print("\nSorry, I don't know", term)
```

Adding a Key-Value Pair

```
elif choice == "2":
    term=input("What term do you want me to add?")
    if term not in geek:
        definition = input("\nWhat's the definition?: ")
        geek[term] = definition
        print("\n", term, "has been added.")
    else:
        print("\nThat term already exists! “,
                  ”Try redefining it.")
```

- **geek[term] = definition** is exactly how you assign a new item to a dictionary: The term is the key and the definition is its value.

Replacing a Key-Value Pair

```
elif choice == "3":
    term =input("What term do you want me to
redefine?: ")
    if term in geek:
        definition = input("What's the new definition?: ")
        geek[term] = definition
        print("\n", term, "has been redefined.")
    else:
        print("\nThat term doesn't exist!  Try adding it.")
```

- To replace a key-value pair, we use the exact same line that we used for adding a new pair: geek[term] = definition.

- If you assign a value to a dictionary using a key that already exists, Python replaces the current value without complaint. So you have to watch out, because you might overwrite the value of an existing key without realizing it.

Deleting a Key-Value Pair

```
elif choice == "4":
    term=input("What term do you want me to delete? ")
    if term in geek:
        del geek[term]
        print("\nOkay, I deleted", term)
    else:
        print("\nI can't do that!", term,
                  "doesn't exist in the dictionary.")
```

- The program asks the user for the geek term to delete. Next, the program checks if the term is in the dictionary. If it is, the item is deleted with

del geek[term]

- This deletes the item with the key term from the dictionary geek.

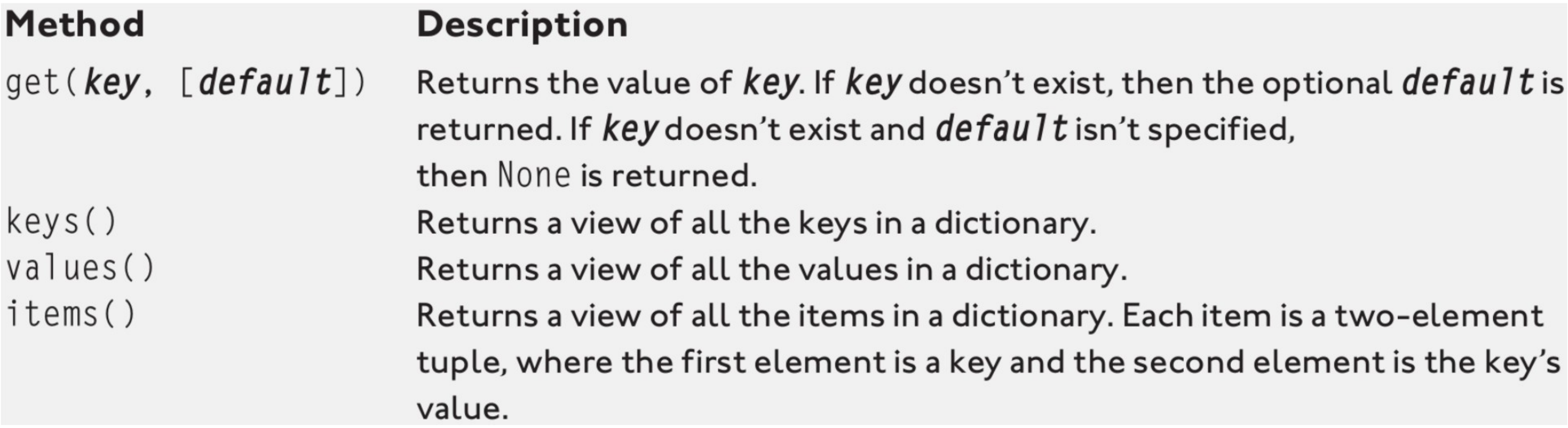
Understanding Dictionary Requirements

- For creating dictionaries:

* A dictionary can't contain multiple items with the same key.

* A key has to be immutable. It can be a string, a number, or a tuple, which gives you lots of possibilities.

* Values don't have to be unique. Also, values can be mutable or immutable. They can be anything you want.



Method	Description
`get(`*`key`*`, [`*`default`*`])`	Returns the value of *key*. If *key* doesn't exist, then the optional *default* is returned. If *key* doesn't exist and *default* isn't specified, then `None` is returned.
`keys()`	Returns a view of all the keys in a dictionary.
`values()`	Returns a view of all the values in a dictionary.
`items()`	Returns a view of all the items in a dictionary. Each item is a two-element tuple, where the first element is a key and the second element is the key's value.

hangman.py

```
# Hangman Game
#
# The classic game of Hangman.  The computer picks a
# random word and the player wrong to guess it, one
# letter at a time.  If the player can't guess the word in
# time, the little stick figure gets hanged.

# imports

import random

# constants
HANGMAN = (
"""
```

```
 ------
 |    |
 |
 |
 |
 |
 |
 |
 |
----------
""",
"""

 ------
 |    |
 |   O
 |
 |
 |
 |
 |
 |
----------
```

```
""",
"""

 ------
 |    |
 |   O
 |  -+-
 |
 |
 |
 |
 |
----------
""",
"""
```

```
 ------
 |    |
 |   O
 | /-+-
 |
 |
 |
 |
 |
----------
""",
"""

 ------
 |    |
 |   O
 | /-+-/
 |
 |
 |
 |
 |
----------
```

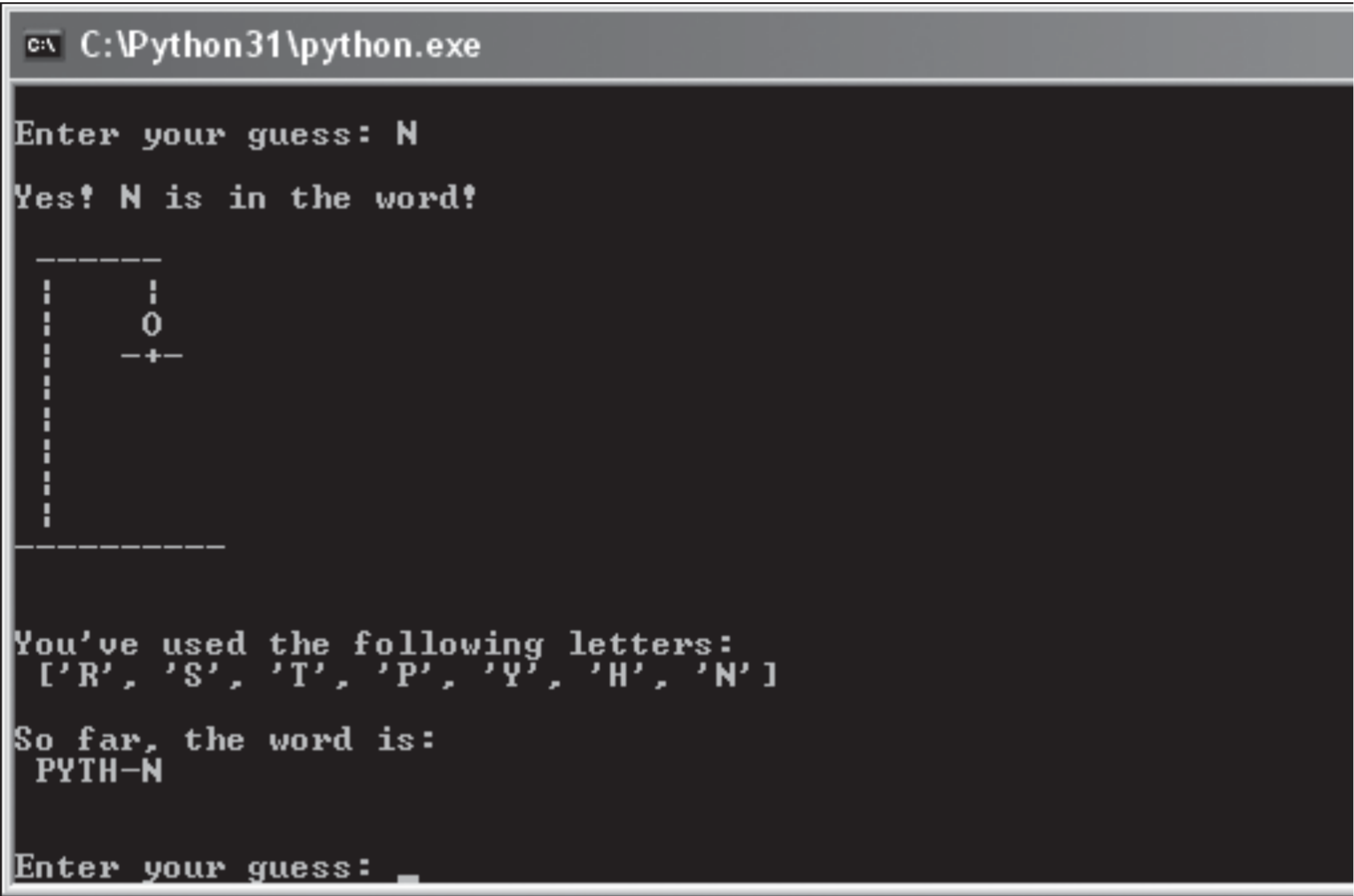
```
""",
"""

 ------
 |    |
 |    O
 | /-+-/
 |    |
 |
 |
 |
 |
----------
""",
"""
```

```
 ------
 |   |
 |   O
 | /-+-/
 |   |
 |   |
 |  |
 |  |
 |
----------
""",
"""

 ------
 |   |
 |    O
 |  /-+-/
 |   |
 |   |
 |  |  |
 |  |  |
 |
----------
```

```
""")

MAX_WRONG = len(HANGMAN) - 1
WORDS = ("OVERUSED", "CLAM", "GUAM",
                  "TAFFETA", "PYTHON")

# initialize variables
# the word to be guessed
word = random.choice(WORDS)

# one dash for each letter in word to be guessed
so_far = "-" * len(word)

# number of wrong guesses player has made
wrong = 0
used = []             # letters already guessed

print("Welcome to Hangman.  Good luck!")
```

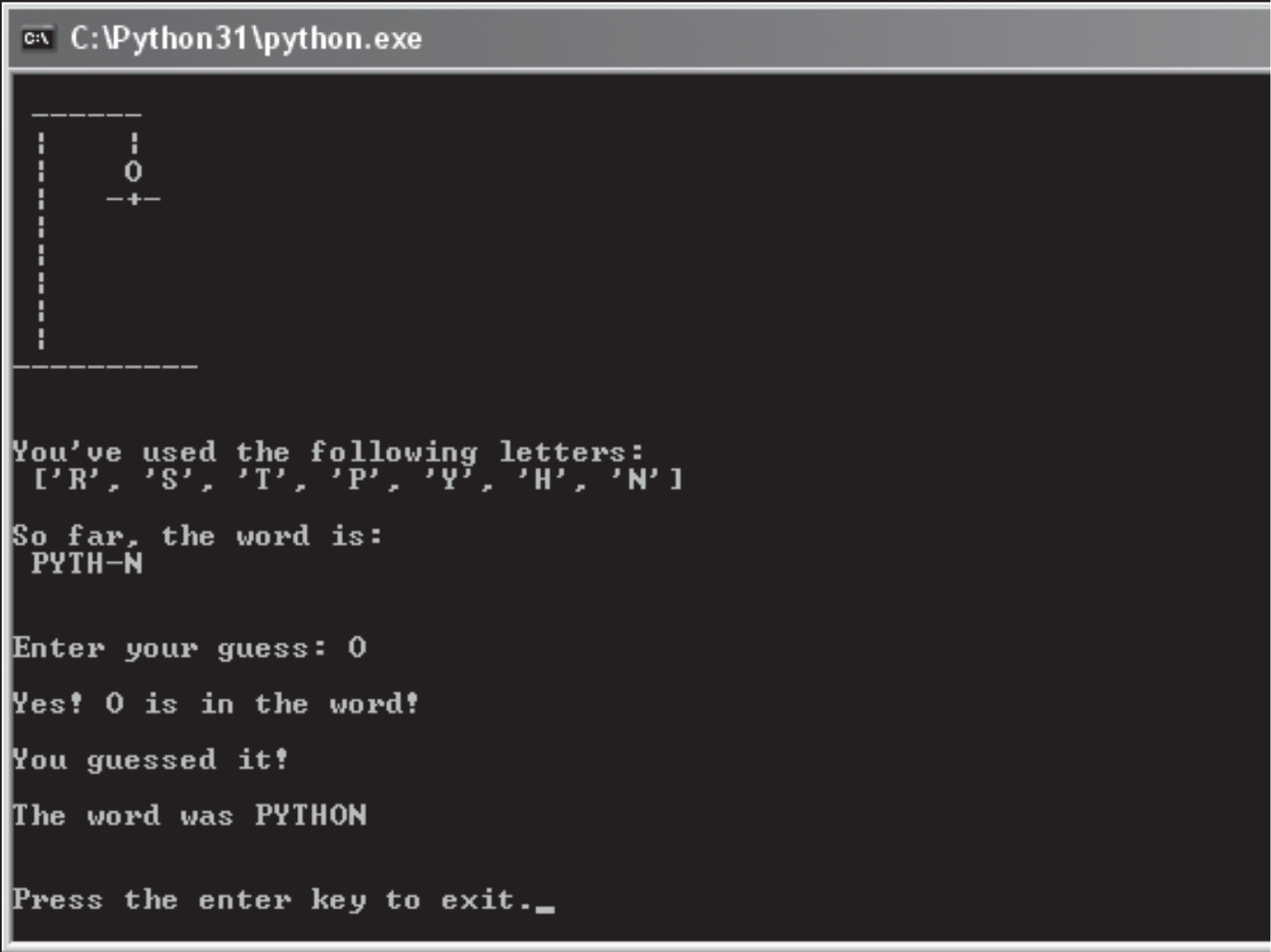
```python
while wrong < MAX_WRONG and so_far != word:
    print(HANGMAN[wrong])
    print("\nYou've used the following letters:\n", used)
    print("\nSo far, the word is:\n", so_far)

    guess = input("\n\nEnter your guess: ")
    guess = guess.upper()

    while guess in used:
        print("You've already guessed the letter", guess)
        guess = input("Enter your guess: ")
        guess = guess.upper()

    used.append(guess)

    if guess in word:
        print("\nYes!", guess, "is in the word!")

        # create a new so_far to include guess
        new = ""
        for i in range(len(word)):
            if guess == word[i]:
                new += guess
            else:
                new += so_far[i]
        so_far = new

    else:
        print("\nSorry,", guess,
                 "isn't in the word.")
        wrong += 1

if wrong == MAX_WRONG:
    print(HANGMAN[wrong])
    print("\nYou've been hanged!")
else:
    print("\nYou guessed it!")

print("\nThe word was", word)

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

Introducing the Hangman Game

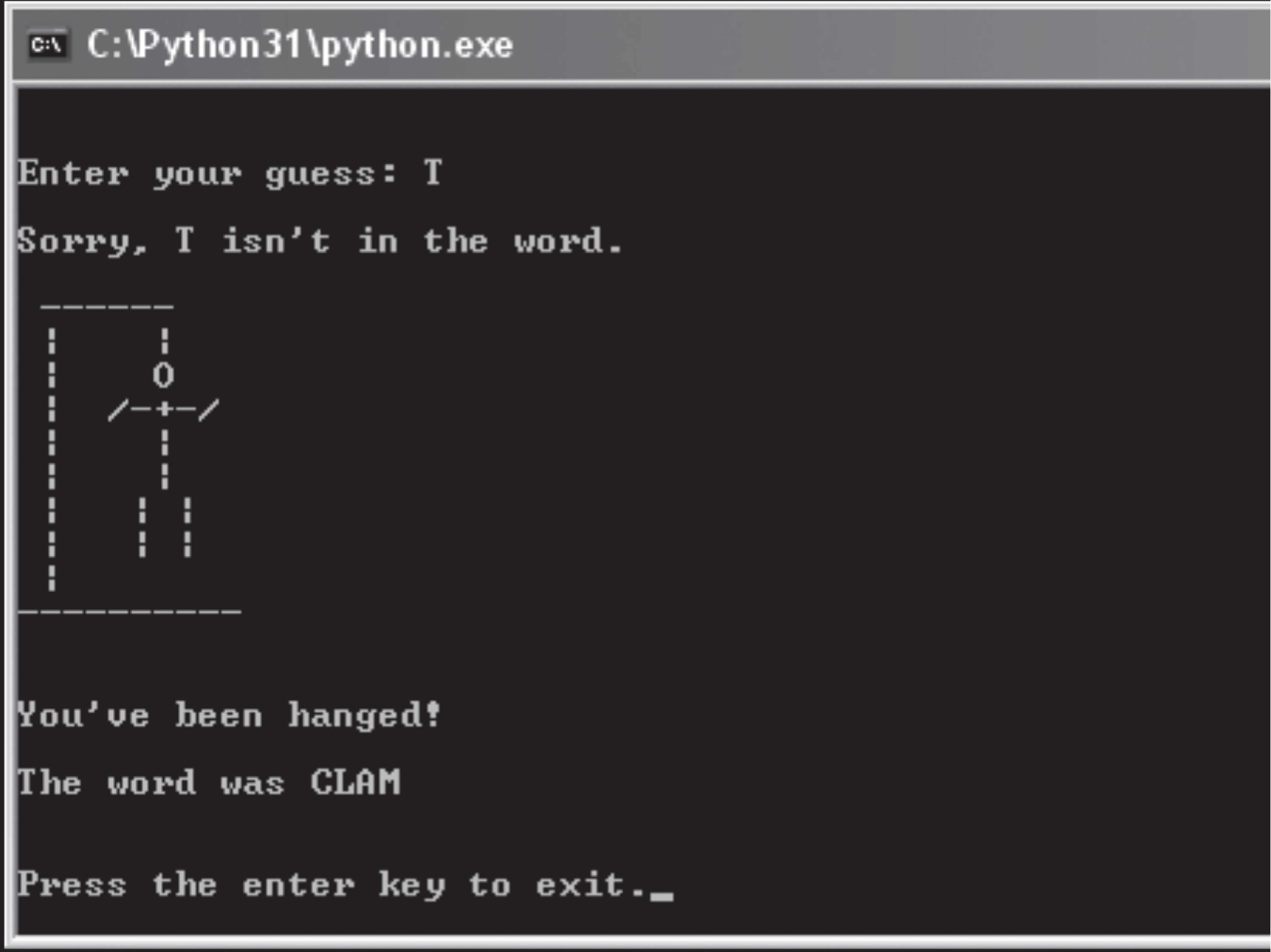


C:\Python31\python.exe

```
Enter your guess: N

Yes! N is in the word!

  ------
  |    |
  |    O
  |   -+-
  |
  |
  |
  |
  |
----------


You've used the following letters:
 ['R', 'S', 'T', 'P', 'Y', 'H', 'N']

So far, the word is:
 PYTH-N


Enter your guess: _
```



```
 ------
 |    |
 |    O
 |   -+-
 |
 |
 |
 |
 |
----------


You've used the following letters:
 ['R', 'S', 'T', 'P', 'Y', 'H', 'N']

So far, the word is:
 PYTH-N


Enter your guess: O

Yes! O is in the word!

You guessed it!

The word was PYTHON


Press the enter key to exit._
```



```
Enter your guess: T

Sorry, T isn't in the word.

  ------
  |    |
  |    O
  |  /-+-/
  |    |
  |    |
  |   | |
  |   | |
  |
----------


You've been hanged!

The word was CLAM


Press the enter key to exit._
```

Creating Constants

- We create the tuple which is a sequence of 8 elements, but each element is a triple-quoted string that spans 12 lines.

- Each string is a representation of the gallows where the stick figure is being hanged. Each subsequent string shows a more complete figure.

- Each time the player guesses incorrectly, the next string is displayed. By the 8 entry, the image is complete and the figure is a goner.

- If this final string is displayed, the player has lost and the game is over.

- We assign this tuple to HANGMAN , a variable name in all caps, because we will be using it as a constant.

```
 ------
 |    |
 |
 |
 |
 |
 |
 |
 |
----------
```

```
 ------
 |    |
 |    0
 |
 |
 |
 |
 |
 |
----------
```

```
 ------
 |    |
 |    0
 |   -+-
 |
 |
 |
 |
 |
----------
```

```
 ------
 |    |
 |    0
 |  /-+-
 |
 |
 |
 |
 |
----------
```

```
 ------
 |    |
 |    0
 |  /-+-/
 |
 |
 |
 |
 |
----------
```

```
 ------
 |    |
 |    0
 |  /-+-/
 |    |
 |
 |
 |
 |
----------
```

```
 ------
 |    |
 |    0
 |  /-+-/
 |    |
 |    |
 |   |
 |   |
 |
----------
```

```
 ------
 |    |
 |    0
 |  /-+-/
 |    |
 |    |
 |   | |
 |   | |
 |
----------
```

- Next, we create a constant to represent the maximum number of wrong guesses a player can make before the game is over:

```
MAX_WRONG = len(HANGMAN) – 1
```

- Finally, we create a tuple containing all of the possible words that the computer can pick from for the player to guess:

```
WORDS = ("OVERUSED", "CLAM", "GUAM",
"TAFFETA", "PYTHON")
```

Initializing the Variables

- We use the random.**choice()** function to pick a random word from the list of possible words:

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

- We create another string, so_far, to represent what the player has guessed so far in the game. so_far starts out as just a series of dashes, one for each letter in the word:

```
so_far = "-" * len(word)
```

- When the player correctly guesses a letter, the dashes in the positions of that letter are replaced with the letter itself.

- We create wrong to keep track of the number of wrong guesses the player makes.

- We create an empty list, used, to contain all the letters the player has guessed: used = []

Creating the Main Loop

- We create a loop that continues until either the player has guessed too many wrong letters or the player has guessed all the letters in the word:

```
while wrong < MAX_WRONG and so_far != word:
    print(HANGMAN[wrong])
    print("\nYou've used the following letters:\n", used)
    print("\nSo far, the word is:\n", so_far)
```

- And we print the current stick figure, based on the number of wrong guesses the player has made.

- Then we display the list of letters that the player has used in this game. And then we show what the partially guessed word looks like so far.

Getting the Player’s Guess

```
guess = input("\n\nEnter your guess: ")
guess = guess.upper()

while guess in used:
    print("You've already guessed the letter", guess)
    guess = input("Enter your guess: ")
    guess = guess.upper()

used.append(guess)
```

Checking the Guess

```
if guess in word:
    print("\nYes!", guess, "is in the word!")

    # create a new so_far to include guess
    new = ""
    for i in range(len(word)):
        if guess == word[i]:
            new += guess
        else:
            new += so_far[i]
    so_far = new

else:
    print("\nSorry,", guess, "isn't in the word.")
    wrong += 1
```

Ending the Game

```
if wrong == MAX_WRONG:
    print(HANGMAN[wrong])
    print("\nYou've been hanged!")
else:
    print("\nYou guessed it!")

print("\nThe word was", word)

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

Quiz 5: Write a Character Creator program for a role-playing game. The player should be given a pool of 30 points to spend on 4 attributes: Strength, Health, Wisdom, and Dexterity. The player should be able to spend points from the pool on any attribute and should also be able to take points from an attribute and put them back into the pool.