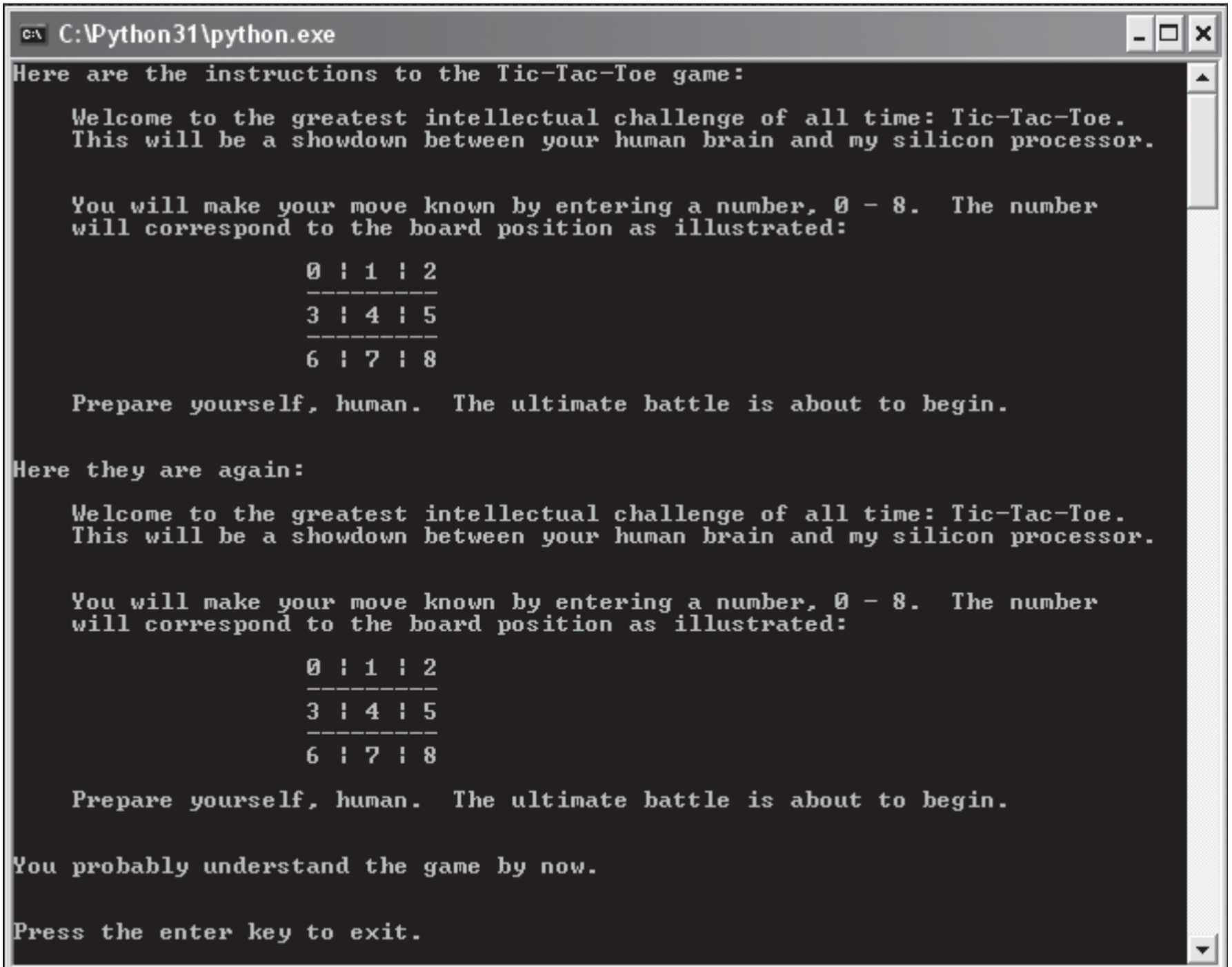
Chapter 6

Functions:
The Tic-Tac-Toe Game

Introducing the Instructions Program



instructions.py

```
# Demonstrates programmer-created functions

def instructions():
    """Display game instructions."""
    print(
    """
    Welcome to the greatest challenge of all time:
    Tic-Tac-Toe. This is a showdown between your human
    brain and my silicon processor.

    You make your move known by entering a number,
    0 - 8.  The number will correspond to the board
    position as illustrated:
                                    0 | 1 | 2
                                    ----------
                                    3 | 4 | 5
                                    ----------
                                    6 | 7 | 8
```

```
    Prepare yourself, human.  The ultimate battle is
    about to begin. \n
    """
    )

# main
print("Here are the instructions to the Tic-Tac-Toe \
game:")
instructions()
print("Here they are again:")
instructions()
print("You probably understand the game by now.")

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

Defining a Function

- the definition of a new function

def instructions():

- This line tells the computer that the block of code that follows is to be used together as the function instructions().

- This means that when we call the function instructions() in this program, the block of code runs.

- This line and its block are a *function definition*. They define what the function does, but don't run the function. The computer won't actually run the function until it sees a function call for it.

- To define a function: start with **def**, followed by a function name, followed by a pair of parentheses, followed by a colon, and then the indented block of statements.

Documenting a Function

- Functions allow you to document them with what's called a *docstring* (or *documentation string*):

```
"""Display game instructions."""
```

- A docstring is typically a triple-quoted string and, if you use one, must be the 1st line in your function.

- Functions work just fine without docstrings, but using them is a good idea.

- It gets you in the habit of commenting your code and makes you describe the function's one, well-defined job.

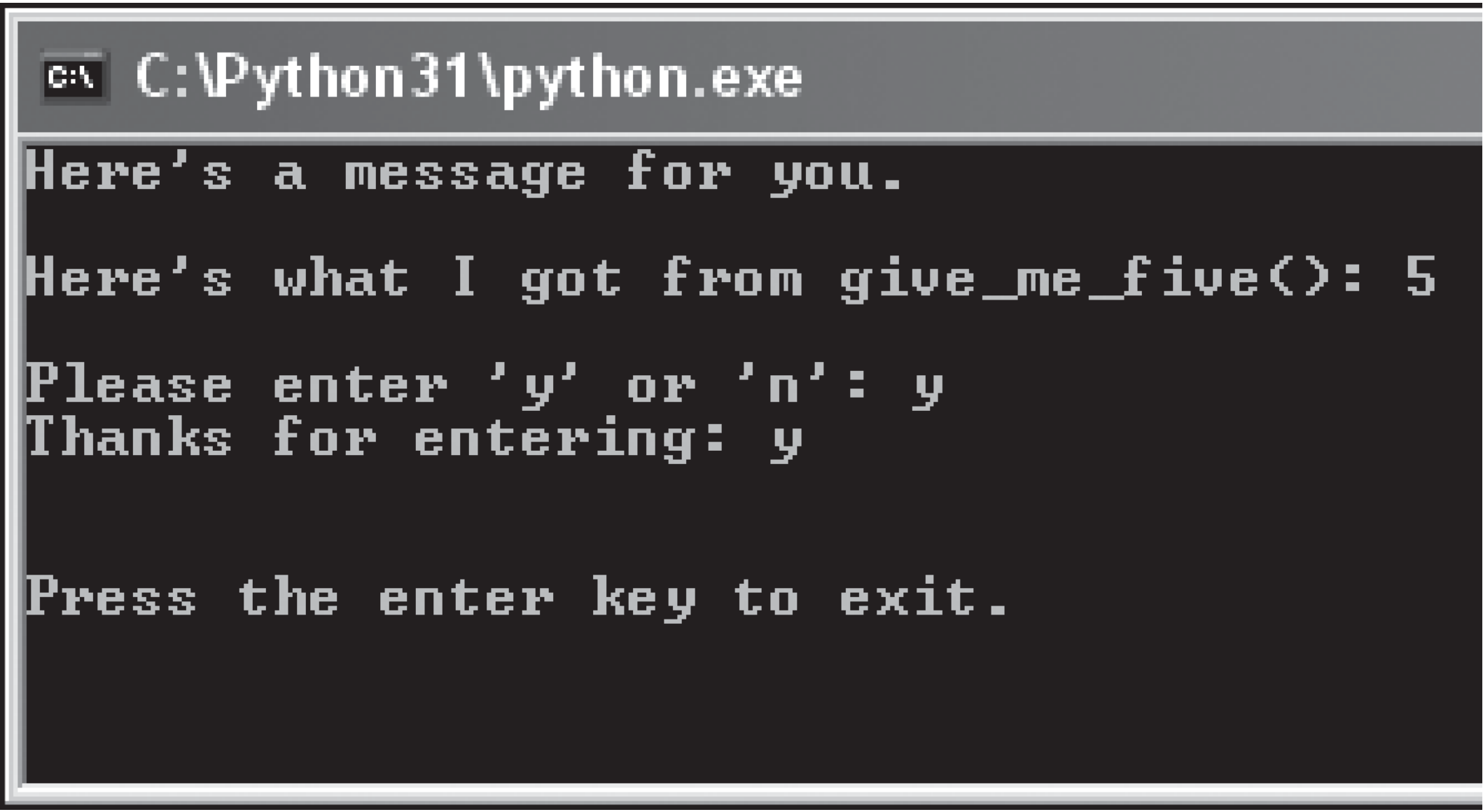
Calling a Programmer-Created Function

- Use the name of the function followed by a set of parentheses:

instructions()

- This tells the computer to go off and execute the function defined earlier.

The Receive and Return Program



receive_and_return.py

```
# Receive and Return
# Demonstrates parameters and return values

def display(message):
    print(message)

def give_me_five():
    five = 5
    return five

def ask_yes_no(question):
    """Ask a yes or no question."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response
```

```
# main
display("Here's a message for you.\n")

number = give_me_five()
print("Here's what I got from give_me_five():", number)

answer = ask_yes_no("\nPlease enter 'y' or 'n': ")
print("Thanks for entering:", answer)

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

Receiving Information through Parameters

- display() receives a value via its *parameter* and prints it:

def display(message):

- Parameters catch the values sent to the function from a function call through its arguments.

- When display() is called, message is assigned the value provided via the argument "Here's a message for you.\n"

- If we hadn't passed message a value, we would have gotten an error. display() requires exactly one argument value.

- Functions can have many parameters. To define a function with multiple parameters, list them out, separated by commas.

Returning Information through Return Values

- give_me_five() returns a value via the **return** statement:

```
return five
```

- When this line runs, the function passes the value of five back to the part of the program that called it, and then ends.

- A function always ends after it hits a **return** statement.

- The main part of the program where we call the function:

```
number = give_me_five()
print("Here's what I got from give_me_five():", number)
```

- Catch the return value of the function by assigning the result of the function call to number.

- You can pass more than one value back from a function. Just list all the values you want to return, separated by commas.

- Make sure to have enough variables to catch all the return values of a function. If you don’t have the right number when you try to assign them, you’ll generate an error.

Understanding Encapsulation

- No variable created in a function, including its parameters, can be directly accessed outside its function. This is called *encapsulation*.

- Encapsulation helps keep independent code truly separate by hiding or encapsulating the details.

- Parameters and return values are used to communicate just the information that needs to be exchanged.

- We don’t have to keep track of variables we create within a function in the rest of our program. As the programs get large, this is a great benefit.

Receiving and Returning Values in the Same Function

- ask_yes_no() receives one value and returns another.

def ask_yes_no(question):

- question gets the value of the argument passed to the function. In this case, it's a string.

- Then the function prompts the user for a response:

```
response = None
while response not in ("y", "n"):
    response = input(question).lower()
```

- The while loop keeps asking the question until the user enters either y , Y , n , or N . After the user has entered a valid response, the function sends a string back return response to end the function.

Understanding Software Reuse

- Another great thing about functions is that they can easily be reused in other programs.

- This type of thing is called *software reuse*.

- It's always a waste of time to "reinvent the wheel," so software reuse, using existing software and other project elements in new projects, is a technique that business has taken to heart.

- So writing good functions not only saves you time and energy in your current project, but can also save you effort in future ones!

- We can create our own modules and import our functions into a new program, just like we import standard Python modules and use their functions. (Chapter 9)

- Software reuse can do the following:

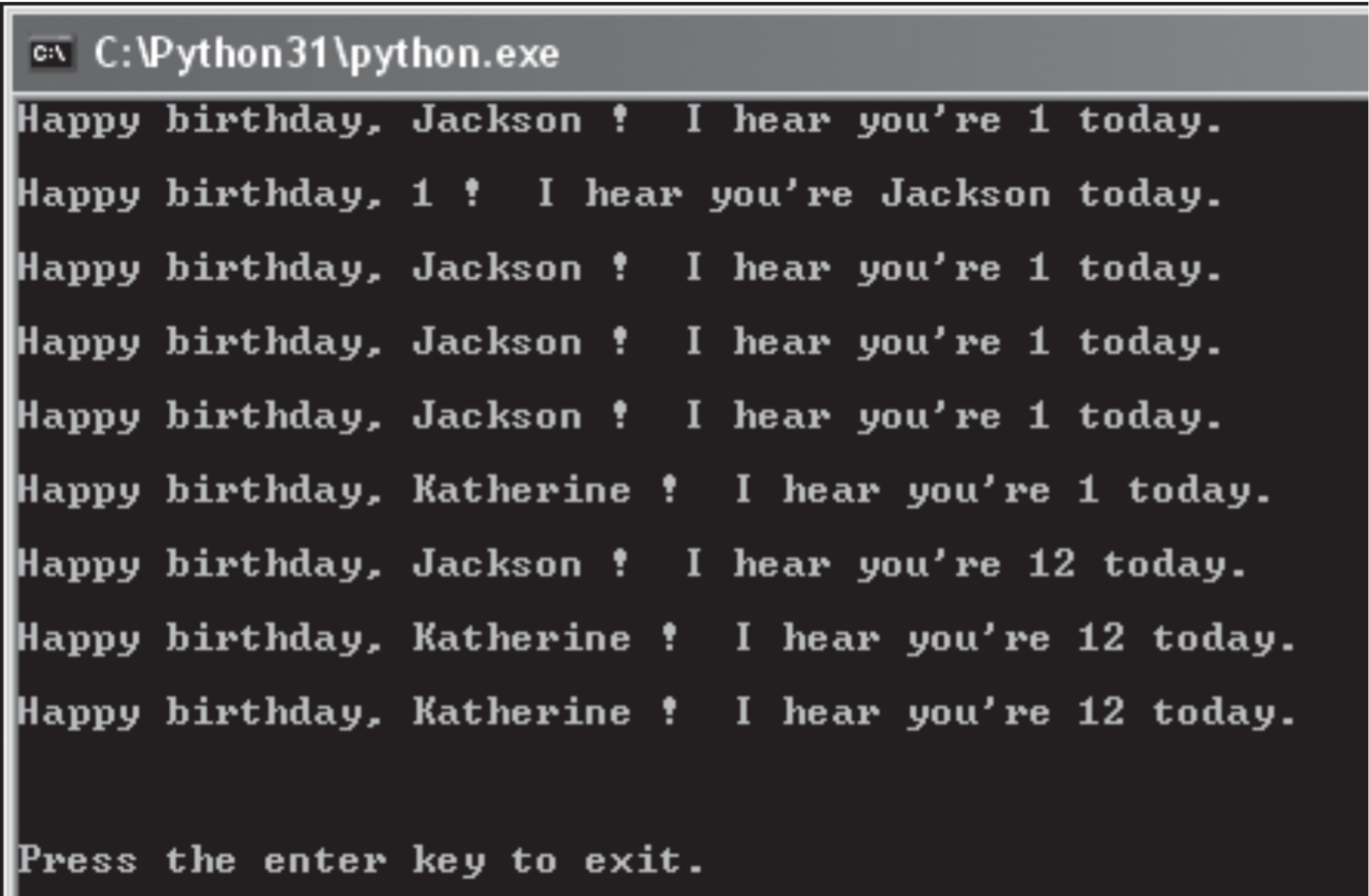
* Increase company productivity.

* Improve software quality.

* Provide consistency across software products.

* Improve software performance.

Introducing the Birthday Wishes Program



birthday_wishes.py

```
# Birthday Wishes: Demonstrates keyword arguments
# and default parameter values

# positional parameters
def birthday1(name, age):
    print("Happy birthday,", name, "!", " I hear you're",
    age, "today.\n")

# parameters with default values
def birthday2(name = "Jackson", age = 1):
    print("Happy birthday,", name, "!", " I hear you're",
    age, "today.\n")

birthday1("Jackson", 1)
birthday1(1, "Jackson")
birthday1(name = "Jackson", age = 1)
birthday1(age = 1, name = "Jackson")
```

```
birthday2()
birthday2(name = "Katherine")
birthday2(age = 12)
birthday2(name = "Katherine", age = 12)
birthday2("Katherine", 12)

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

Using Positional Parameters and Positional Arguments

- List out a series of variable names in a function's header, you create positional parameters:

def birthday1(name, age):

- If you call a function with just a series of values, you create positional arguments:

birthday1("Jackson", 1)

- Using positional parameters and positional arguments means that parameters get their values based on the position of the values sent. The 1st parameter gets the 1st value sent, the 2nd parameter gets the 2nd value sent, and so on.

- So the result is:

Happy Birthday, Jackson ! I hear you’re 1 today.

- If you switch the positions of 2 arguments, the parameters get different values:

birthday1(1, "Jackson")

- The result becomes:

Happy Birthday, 1 ! I hear you’re Jackson today.

Using Positional Parameters and Keyword Arguments

- You can tell the function to assign certain values to specific parameters, regardless of order, if you use keyword arguments.

- With keyword arguments, you use the actual parameter names from the function header to link a value to a parameters: **birthday1(name = "Jackson", age = 1)**

- The beauty of keyword arguments is that their order doesn't matter; it's the keywords that link values to parameters: **birthday1(age = 1, name = "Jackson")** give the same result.

- Keyword arguments let you pass values in any order. But their biggest benefit is clarity.

Using Default Parameter Values

- You have the option to assign default values to your parameters, values that get assigned to the parameters if no value is passed to them:

def birthday2(name = "Jackson", age = 1):

- This means that if no value is supplied to name, it gets "Jackson". And if no value is supplied for age, it gets 1.

- So the call

birthday2()

doesn’t generate an error; instead, the default values are assigned to the parameters:

- Once you assign a default value to a parameter in the list, you have to assign default values to all the parameters listed **after** it:

```
def monkey(bananas=10, barrel="yes", uncle="your"):
```

√

```
def monkey(bananas=10, barrel, uncle):
```

✗

- You can override the default values of any or all the parameters:

```
birthday2(name = "Katherine")
birthday2(age = 12)
birthday2(name = "Katherine", age = 12)
birthday2("Katherine", 12)
```

- Default parameter values are great if you have a function where almost every time it’s called, some parameter gets sent the same value.

Anonymous Functions: lambda

- Using **lambda** is a easy to write a simple function:

lambda arguments : expression

```
add10 = lambda a : a + 10
print(add10(5))   # 15

axb = lambda a, b : a * b
print(axb(5, 6))   # 30

low=lambda x, y: x if x < y else y
print(lower('cc','bb'))   # ‘bb’
print(lower('aa','bb'))   # 'aa'
```

```
def add10(a):
    return a + 10

def axb(a,b):
    return a * b

def low(x,y):
    if x < y: return x
    else:     return y
```

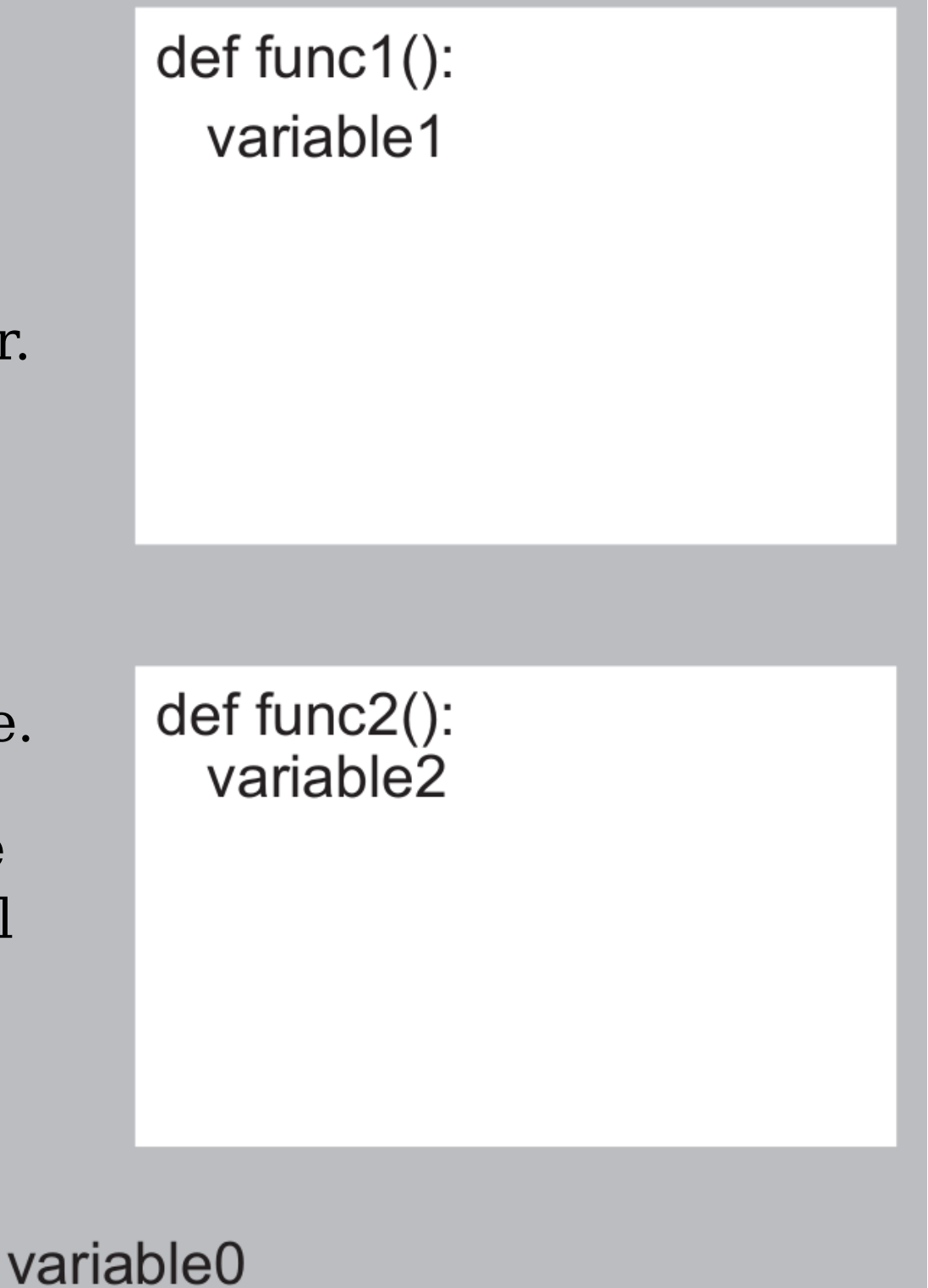
- Use lambda because it is easy and handy;
- Don’t use lambda to avoid messing up your code.

Understanding Scopes

- *Scopes* represent different areas of your program that are separate from each other.

- The 1st scope is defined by func1(), the 2nd is defined by func2(), and the 3rd is the global scope, which all programs automatically have.

- Any variable created in the *global scope* is called a global variable, while any variable created inside a function is called a *local variable* (it's local to that function).

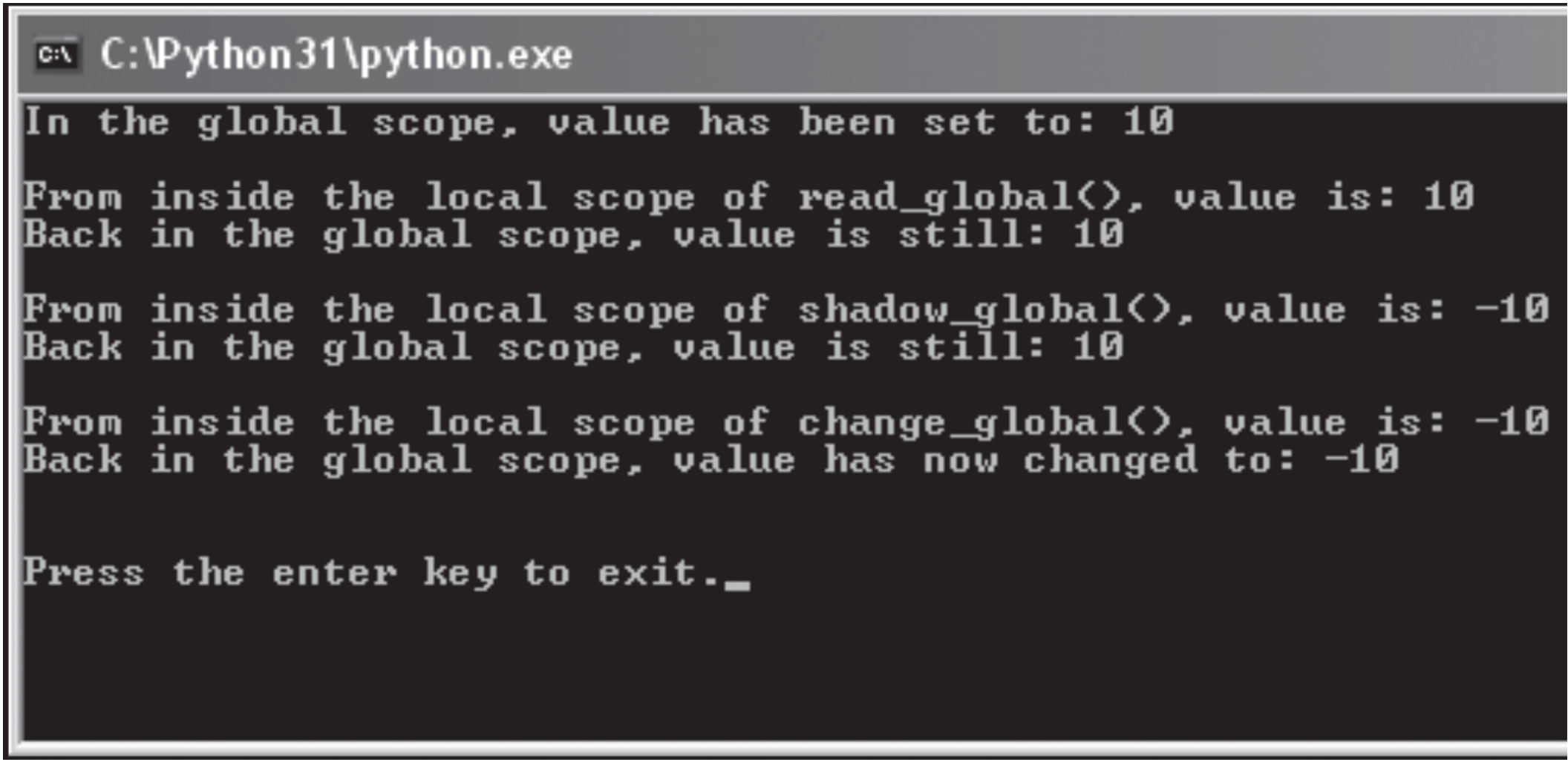


- Since variable1 is defined in func1(), it's a local variable that lives only in the scope of func1(). variable1 can't be accessed from any other scope. So, no command in func2() can get at it, and no command in the global space can access or modify it either.

- When you're in a function, you have access to all of its variables. But when you're outside a function, like in the global scope, you can't see any of the variables inside a function.

- If 2 variables have the same name inside 2 separate functions, they're totally different variables with no connection to each other.

Introducing the Global Reach Program



global_reach.py

```
# Global Reach
# Demonstrates global variables

def read_global():
    print("From inside the local scope of read_global(),”,
    “ value is:", value)

def shadow_global():
    value = -10
    print("From inside the local scope of shadow_global()”,
    “ value is:", value)

def change_global():
    global value
    value = -10
    print("From inside the local scope of change_global()”,
    “ value is:", value)
```

```
# main
# value is a global variable since we're in the global
# scope here

value = 10

print("In the global scope, value has been set to:",
        value, "\n")

read_global()
print("Back in the global scope, value is still:",value,"\n")

shadow_global()
print("Back in the global scope, value is still:",value,"\n")

change_global()
print("Back in the global scope,",
        " value has now changed to:", value)

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

Reading a Global Variable from Inside a Function

- you can read the value of a global variable from within any scope in your program, eg, in the function read_global().

- While you can always read the value of a global variable in any function, you can’t change it directly.

- So, in read_global(), doing something like

value += 1

would generate a nasty error.

Shadowing a Global Variable from Inside a Function

- If you give a variable inside a function the same name as a global variable, you *shadow* the global variable.

- It might look like you can change the value of a global variable by doing this, but you only change the local variable you've created.

- In shadow_global():

value = – 10

This doesn't change the global version of value. Instead, we create a new, local version of value inside the function and that got –10. But he global version of value won't change.

- It's not a good idea to shadow a global variable inside a function. It can lead to confusion.

Changing a Global Variable from Inside a Function

- To gain complete access to a global variable, use the keyword **global** like in change_global():

```
global value
```

- At this point, the function has complete access to value.

- So when we change it with

```
value = – 10
```

the global variable value got –10, no matter inside the function, or back in the main part of the code.

Understanding When to Use Global Variables and Constants

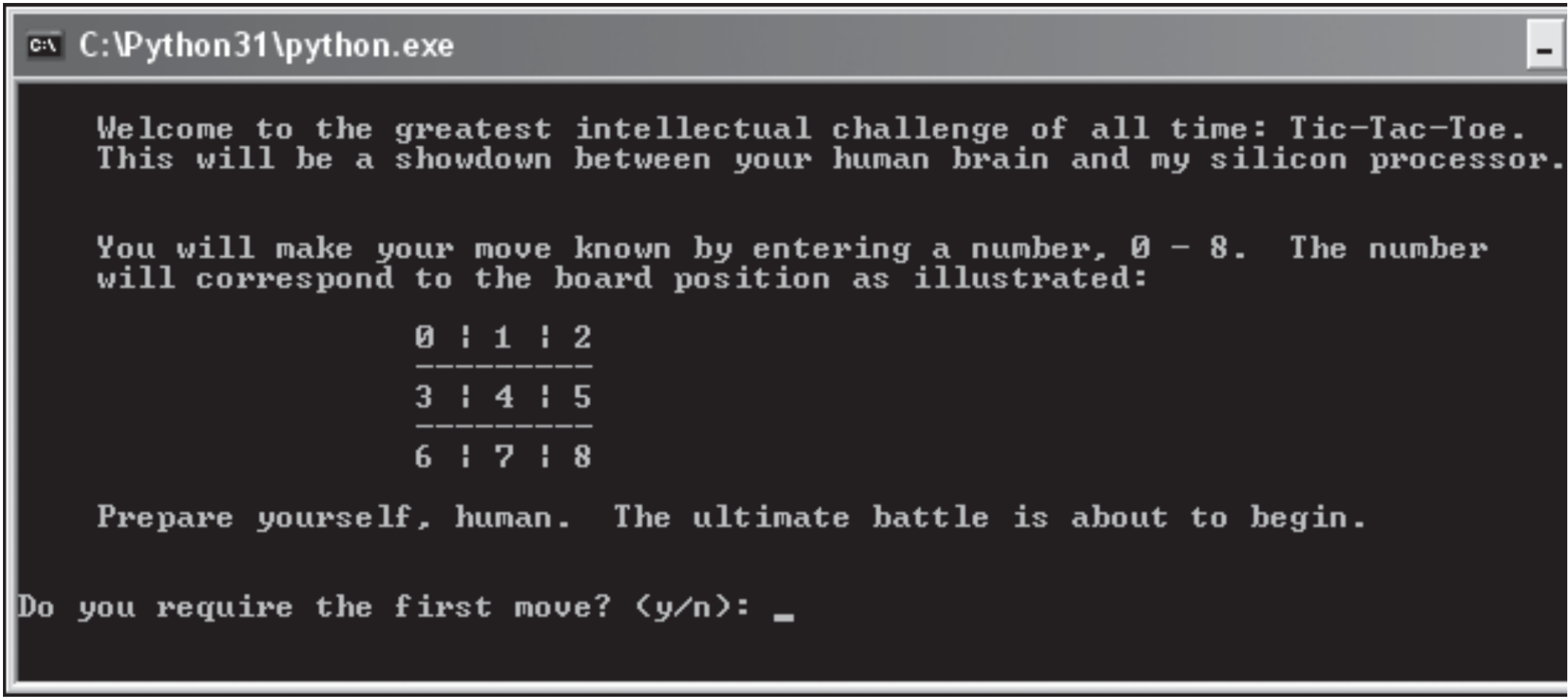
- In general, global variables make programs confusing because it can be hard to keep track of their changing values

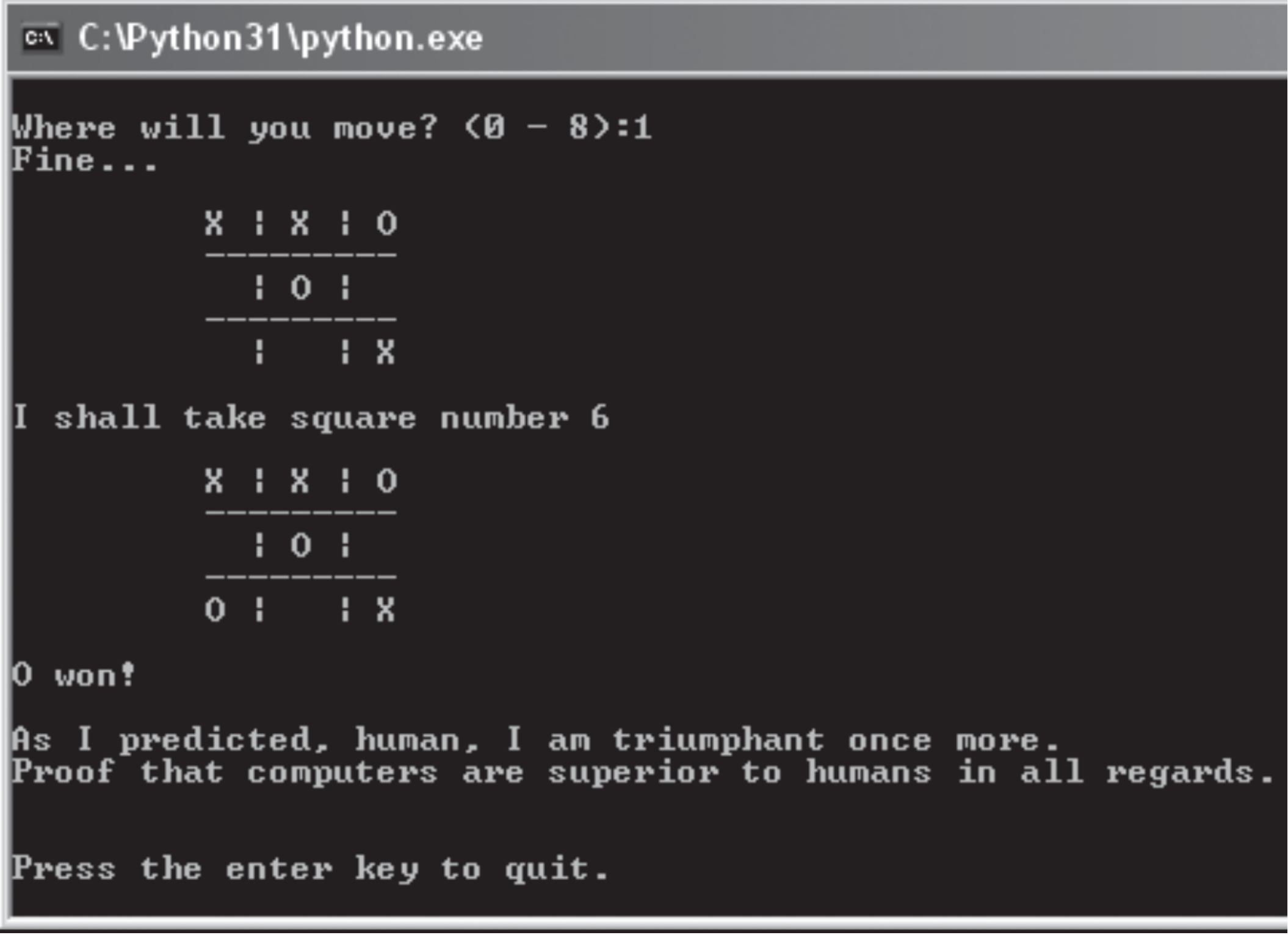
- You should limit your use of them as much as you can.

- Global constants (global variables treated as constants), on the other hand, can make programs less confusing.

- It makes your code clearer and it makes changes no sweat when the global constants change (like gas price).

Introducing Tic-Tac-Toe Game



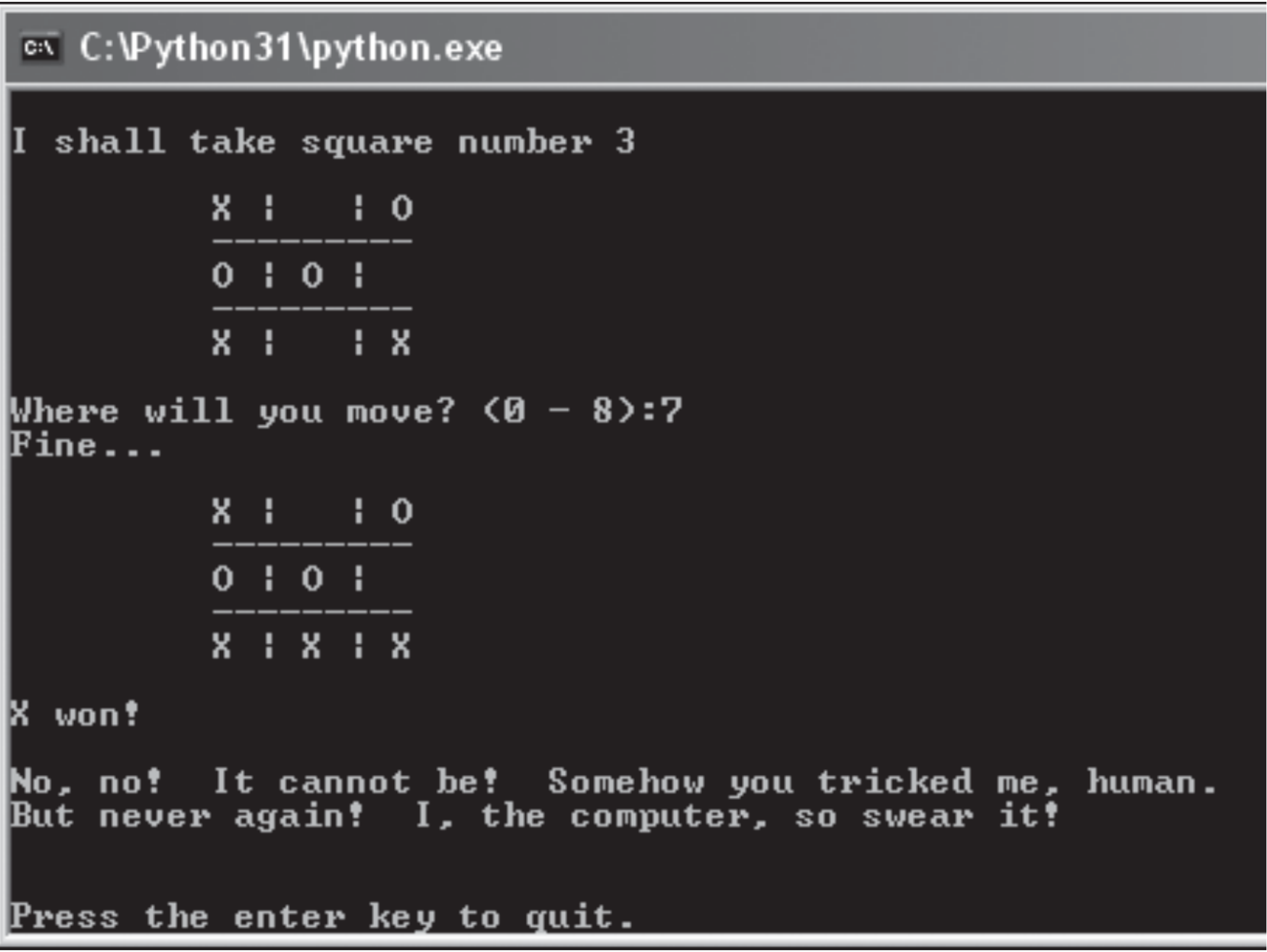

C:\Python31\python.exe
Where will you move? (0 - 8):1
Fine...
X | X | O

| O |

| | X
I shall take square number 6
X | X | O

| O |

O | | X
O won!
As I predicted, human, I am triumphant once more.
Proof that computers are superior to humans in all regards.
Press the enter key to quit.


C:\Python31\python.exe
I shall take square number 3
X | | O

O | O |

X | | X
Where will you move? (0 - 8):7
Fine...
X | | O

O | O |

X | X | X
X won!
No, no! It cannot be! Somehow you tricked me, human.
But never again! I, the computer, so swear it!
Press the enter key to quit.

Planning the Tic-Tac-Toe Game: Writing the Pseudocode

```
display the game instructions
determine who goes first
create an empty tic-tac-toe board
display the board

while nobody's won and it's not a tie
    if it's the human's turn
        get the human's move
        update the board with the move
    otherwise
        calculate the computer's move
        update the board with the move
    display the board
    switch turns

congratulate the winner or declare a tie
```

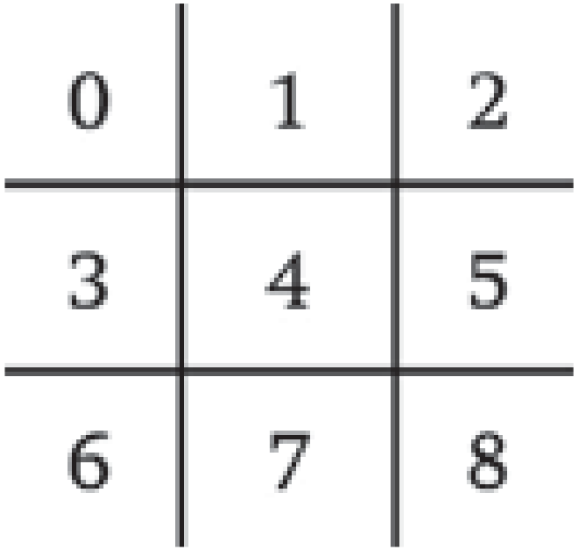
Representing the Data

- Since we are going to print the game board on the screen, we can represent a piece as one character, an "X" or an "O". And an empty piece could just be a space.

- The board itself should be a list since it's going to change as each player makes a move.

- There are 9 squares on a tic-tac-toe board, so the list should be 9 elements long:

- The list will be 9 elements long and have position numbers 0–8.



- The sides the player and computer play could also be represented by "X" and "O". And a variable to represent the side of the current turn would be either an "X" or an "O".

Creating a List of Functions

Function	Description
`display_instruct()`	Displays the game instructions.
`ask_yes_no(`*`question`*`)`	Asks a yes or no question. Receives a question. Returns either a `"y"` or a `"n"`.
`ask_number(`*`question`*`, `*`low`*`, `*`high`*`)`	Asks for a number within a range. Receives a question, a low number, and a high number. Returns a number in the range from *low* to *high*.
`pieces()`	Determines who goes first. Returns the computer's piece and human's piece.
`new_board()`	Creates a new, empty game board. Returns a board.
`display_board(`*`board`*`)`	Displays the board on the screen. Receives a board.
`legal_moves(`*`board`*`)`	Creates a list of legal moves. Receives a board. Returns a list of legal moves.
`winner(`*`board`*`)`	Determines the game winner. Receives a board. Returns a piece, `"TIE"` or `None`.
`human_move(`*`board`*`, `*`human`*`)`	Gets the human's move from the player. Receives a board and the human's piece. Returns the human's move.
`computer_move(`*`board`*`, `*`computer`*`, `*`human`*`)`	Calculates the computer's move. Receives a board, the computer piece, and the human piece. Returns the computer's move.
`next_turn(`*`turn`*`)`	Switches turns based on the current turn. Receives a piece. Returns a piece.
`congrat_winner(`*`the_winner`*`, `*`computer`*`, `*`human`*`)`	Congratulates the winner or declares a tie. Receives the winning piece, the computer's piece, and the human's piece.

```
# Tic-Tac-Toe: Plays the game of tic-tac-toe against a
#                human opponent

# global constants
X = "X"
O = "O"
EMPTY = " "
TIE = "TIE"
NUM_SQUARES = 9

def display_instruct():
    """Display game instructions."""
    print(
    """
    Welcome to the greatest intellectual challenge of all
    time: Tic-Tac-Toe. This will be a showdown between
    your human brain and my silicon processor.

    You will make your move known by entering a number,
    0 – 8. The number will correspond to the board
    position as:
```

tic-tac-toe.py

```
                0 | 1 | 2
                -----------
                3 | 4 | 5
                -----------
                6 | 7 | 8
    Prepare yourself.  The battle is about to begin. \n
    """
    )
def ask_yes_no(question):
    """Ask a yes or no question."""
    response = None
    while response not in ("y", "n"):
        response = input(question).lower()
    return response

def ask_number(question, low, high):
    """Ask for a number within a range."""
    response = None
    while response not in range(low, high):
        response = int(input(question))
    return response
```

```
def pieces():
    """Determine if player or computer goes first."""
    go_first =ask_yes_no(\
    "Do you require the first move? (y/n): ")
    if go_first == "y":
        print("\nThen take the first move.  You will need it.")
        human = X
        computer = O
    else:
        print("\nYour bravery will be your undoing...”,
                “ I will go first.")
        computer = X
        human = O
    return computer, human

def new_board():
    """Create new game board."""
    board = []
    for square in range(NUM_SQUARES):
        board.append(EMPTY)
    return board
```

```
def display_board(board):
    """Display game board on screen."""
    print("\n\t", board[0], "|", board[1], "|", board[2])
    print("\t", "---------")
    print("\t", board[3], "|", board[4], "|", board[5])
    print("\t", "---------")
    print("\t", board[6], "|", board[7], "|", board[8], "\n")


def legal_moves(board):
    """Create list of legal moves."""
    moves = []
    for square in range(NUM_SQUARES):
        if board[square] == EMPTY:
            moves.append(square)
    return moves
```

```
def winner(board):
    """Determine the game winner."""
    WAYS_TO_WIN = ((0, 1, 2),
                   (3, 4, 5),
                   (6, 7, 8),
                   (0, 3, 6),
                   (1, 4, 7),
                   (2, 5, 8),
                   (0, 4, 8),
                   (2, 4, 6))

    for row in WAYS_TO_WIN:
        if board[row[0]]==board[row[1]]==board[row[2]]\
                                                !=EMPTY:
            winner = board[row[0]]
            return winner

        if EMPTY not in board:
            return TIE

        return None
```

```
def human_move(board, human):
    """Get human move."""
    legal = legal_moves(board)
    move = None
    while move not in legal:
        move = ask_number("Where will you move? (0 – 8):",
                          0, NUM_SQUARES)
        if move not in legal:
            print("\nThat square is already occupied, foolish”,
                  “human.  Choose another.\n")
    print("Fine...")
    return move

def computer_move(board, computer, human):
    """Make computer move."""
    # make a copy to work with since function changes list
    board = board[:]
    # the best positions to have, in order
    BEST_MOVES = (4, 0, 2, 6, 8, 1, 3, 5, 7)

    print("I shall take square number", end=" ")
```

```
# if computer can win, take that move
for move in legal_moves(board):
    board[move] = computer
    if winner(board) == computer:
        print(move)
        return move
    board[move] = EMPTY      # done checking, undo it

# if human can win, block that move
for move in legal_moves(board):
    board[move] = human
    if winner(board) == human:
        print(move)
        return move
    board[move] = EMPTY      # done checking, undo it

# None can win on next move, pick best open square
for move in BEST_MOVES:
    if move in legal_moves(board):
        print(move)
        return move
```

```
def next_turn(turn):
    """Switch turns."""
    if turn == X:
        return O
    else:
        return X

def congrat_winner(the_winner, computer, human):
    """Congratulate the winner."""
    if the_winner != TIE:
        print(the_winner, "won!\n")
    else:
        print("It's a tie!\n")

    if the_winner == computer:
        print("As I predicted, human, I am triumphant”,
        “ again. \nProof that computers are superior to”,
        “ humans in all regards.")
```

```
    elif the_winner == human:
        print("No!  It cannot be! You tricked me, human.”,
        “ \nBut never again!  I, the computer, so swear it!")

    elif the_winner == TIE:
        print("You were most lucky, human, and somehow”,
        “managed to tie me.  \n", "Celebrate today... for”,
        “ this is the best you will ever achieve.")

def main():
    display_instruct()
    computer, human = pieces()
    turn = X
    board = new_board()
    display_board(board)
```

```
    while not winner(board):
        if turn == human:
            move = human_move(board, human)
            board[move] = human
        else:
            move = computer_move(board, computer, human)
            board[move] = computer
        display_board(board)
        turn = next_turn(turn)

    the_winner = winner(board)
    congrat_winner(the_winner, computer, human)

# start the program
main()
input("\n\nPress the enter key to quit.")
```

Setting Up the Program

- We set up some global constants because these are values that more than one function will use:

```
# global constants
X = "X"
O = "O"
EMPTY = " "
TIE = "TIE"
NUM_SQUARES = 9
```

The ask_number() Function

```
def ask_number(question, low, high):

    """Ask for a number within a range."""

    response = None

    while response not in range(low, high):
        response = int(input(question))

    return response
```

The pieces() Function

```
def pieces():

    """Determine if player or computer goes first."""

    go_first=ask_yes_no(\
    "Do you require the 1st move? (y/n):")

    if go_first == "y":
        print("\nThen take the 1st move.  You will need it.")
        human = X
        computer = O
    else:
        print("\nYour bravery will be your undoing...",
                " I will go first.")
        computer = X
        human = O

    return computer, human
```

The new_board() Function

- This function creates a new board (list) with all 9 elements set to EMPTY and returns it:

```
def new_board():

    """Create new game board."""

    board = []

    for square in range(NUM_SQUARES):
        board.append(EMPTY)

    return board
```

The display_board() Function

- This function displays the board passed to it:

```
def display_board(board):

    """Display game board on screen."""

    print("\n\t", board[0], "|", board[1], "|", board[2])

    print("\t", "---------")

    print("\t", board[3], "|", board[4], "|", board[5])

    print("\t", "---------")

    print("\t", board[6], "|", board[7], "|", board[8], "\n")
```

The legal_moves() Function

- This function receives a board and returns a list of legal moves. And a legal move is represented by the number of an empty square.

- So, this function just loops over the list representing the board. Each time it finds an empty square, it adds that square number to the list of legal moves. Then it returns the list of legal moves:

```
def legal_moves(board):
    """Create list of legal moves."""
    moves = []

    for square in range(NUM_SQUARES):
        if board[square] == EMPTY:
            moves.append(square)

    return moves
```

The winner() Function

- winner() receives a board and returns the winner.

- There are 4 possible values for a winner. winner() will return either X or O if one of the players has won. If every square is filled and no one has won, it returns TIE. If no one has won and there is at least one empty square, the function returns None.

- The constant WAYS_TO_WIN is defined to represents all 8 ways to get 3 in a row. Each way to win is represented by a tuple:

WAYS_TO_WIN=((0, 1, 2),(3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7),(2, 5, 8), (0, 4, 8), (2, 4, 6))

- Next, we use a for loop to go through each possible way a player can win, to see if either player has 3 in a row.

- The if statement checks if the 3 squares in question all contain the same value and are not empty. If so, that means that the row has either 3 X's or O's and somebody has won:

```
for row in WAYS_TO_WIN:
    if board[row[0]] == board[row[1]] \
                        == board[row[2]] != EMPTY:
        winner = board[row[0]]
        return winner
```

- If neither player has won, then the function continues. Next, it checks to see if there are any empty squares left on the board. If there aren't any, the game is a tie:

```
if EMPTY not in board:
    return TIE
```

- If it isn't a tie, the function continues. If neither player has won and the game isn't a tie, there is no winner yet. So, the function returns None: **return None**

The human_move() Function

- human_move() receives a board and the human's piece. It returns the square number where the player wants to move.

- First, the function gets a list of all the legal moves for this board. Then, it asks the user for the square number to which he wants to move:

```
def human_move(board, human):
    """Get human move."""
    legal = legal_moves(board)
    move = None
    while move not in legal:
        move = ask_number("Where will you move?”,
                          “ (0-8):", 0, NUM_SQUARES)
        if move not in legal:
            print("\nThat square is already occupied,”,
                          “ foolish human.  Choose another.\n")
    print("Fine...")
    return move
```

The computer_move() Function

- computer_move() receives the board, the computer's piece, and the human's piece. It returns the computer's move.

- When you get a mutable value passed to a function, eg, board, you have to be careful. If you know you're going to change the value as you work with it, make a copy and use that instead:

```
def computer_move(board, computer, human):
    """Make computer move."""
    # make a copy since function changes list
    board = board[:]
```

The basic strategy for the computer:

1. If there's a move that allows the computer to win this turn, the computer should choose that move.

2. If there's a move that allows the human to win next turn, the computer should choose that move.

3. Otherwise, the computer should choose the best empty square as its move. The best square is the center. The next best squares are the corners. And the next best squares are the rest.

```
# the best positions to have, in order
BEST_MOVES = (4, 0, 2, 6, 8, 1, 3, 5, 7)
```

- After creating a list of all the legal moves, we try the computer's piece in each empty square number from the legal moves list and check for a win.

- If the computer can win, the function returns that move and ends. Otherwise, we undo the move just tried and try the next one in the list:

```
# if computer can win, take that move
for move in legal_moves(board):
    board[move] = computer
    if winner(board) == computer:
        print(move)
        return move

    # done checking the move, undo it
    board[move] = EMPTY
```

- Then we check if the player can win on his next move. If the human can win, then that's the move to take for a block.

- If this is the case, the function returns the move and ends. Otherwise, we undo the move and try the next legal move in the list:

```
# if human can win, block that move

for move in legal_moves(board):
    board[move] = human
    if winner(board) == human:
        print(move)
        return move

    # done checking the move, undo it
    board[move] = EMPTY
```

- Then we look through the list of best moves and take the 1st legal one. The computer loops through BEST_MOVES, and as soon as it finds one that's legal, it returns that move:

```
# None can win on next move, pick best open square

for move in BEST_MOVES:
    if move in legal_moves(board):
        print(move)
        return move
```

The next_turn() Function

- This function receives the current turn and returns the next turn:

```
def next_turn(turn):

    """Switch turns."""

    if turn == X:
        return O
    else:
        return X
```

The main() Function

- Here we put the main part of the program into its own function, instead of leaving it at the global level. This encapsulates the main code too.

- It's usually a good idea to encapsulate even the main part of it.

- The main part of the program is almost exactly, line for line, the pseudocode earlier:

```
def main():
    display_instruct()
    computer, human = pieces()
    turn = X
    board = new_board()
    display_board(board)
```

```
while not winner(board):
    if turn == human:
        move = human_move(board, human)
        board[move] = human
    else:
        move = computer_move(board, computer, human)
        board[move] = computer
    display_board(board)
    turn = next_turn(turn)

the_winner = winner(board)
congrat_winner(the_winner, computer, human)
```

Starting the Program

- The next line calls the main function (which in turn calls the other functions) from the global level:

```
# start the program

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