FACULTY OF COMPUTING
& INFORMATION TECHNOLOGY
KING ABDULAZIZ UNIVERSITY

FCIT
KAU

كلية الحاسبات
وتقنية المعلومات
جامعة الملك عبدالعزيز

Chapter 5
Loops

CPIT 110 (Problem-Solving and Programming)

Version 1.0

Sections

Programs

- Program 1: Subtraction Quiz
- Program 2: Guessing Game
- Program 3: Multiple Subtraction Quiz
- Program 4: Advanced Multiple Subtraction Quiz
- Program 5: Sentinel Value
- Program 6: Multiplication Table
- Program 7: Finding the GCD
- Program 8: Predicting The Future Tuition
- Program 9: Prime Number

Check Points

- Section 5.2
  - #1
  - #2
  - #3
  - #4
- Section 5.3
  - #5
  - #6
  - #7
  - #8
  - #9
- Section 5.4
  - #10
  - #11
  - #12
  - #13
  - #14
  - #15
  - #16
- Section 5.7
  - #17
  - #18
  - #19
  - #20
  - #21
  - #22

Objectives

- To write programs for executing statements repeatedly by using a while loop (5.2).
- To develop loops following the loop design strategy (5.2.1-5.2.3).
- To control a loop with the user’s confirmation (5.2.4).
- To control a loop with a sentinel value (5.2.5).
- To use for loops to implement counter-controlled loops (5.3).
- To write nested loops (5.4).
- To learn the techniques for minimizing numerical errors (5.5).
- To learn loops from a variety of examples (GCD, FutureTuition, MonteCarloSimulation, PrimeNumber) (5.6, 5.8).
- To implement program control with break and continue (5.7).

5.1. Motivations

- Loops

Motivations

- Suppose that you need to display a string (e.g., Programming is fun!) 100 times.
- It would be tedious to type the statement 100 times:

```
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")
...
print("Programming is fun!")
print("Programming is fun!")
print("Programming is fun!")
```

100 times

- **So, how do you solve this problem?**

Motivations

- Solution:
  - Python provides a powerful construct called a loop.
  - A loop controls how many times an operation (or a sequence of operations) is performed.
  - By using a loop statement, you don't have to code the print statement a hundred times.
  - You simply tell the computer to display a string that number of times.
  - The loop statement can be written as follows:

```python
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

The loop statement

Motivations

- Solution:

```python
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

The loop body

- Details:
  - The variable count is initially **0**.
  - The loop checks whether count < **100** is True.
    - If so, the loop body is executed.
      - "Programming is fun!" is printed.
      - Then, count is incremented by **1** (count = count + **1**).
  - When count < **100** is False, the loop will terminate
    - and the next statement after the loop statement is executed.

Loops

- A loop is a construct that controls the repeated execution of a block of statements.
- The concept of looping is fundamental to programming.
- Python provides two types of loop statements:
  - while loops
    - The while loops is a condition-controlled loop.
    - it is controlled by a True/False condition.
  - for loops
    - The for loop is a count-controlled loop.
    - It repeats a specified number of times.

5.2. The while Loop

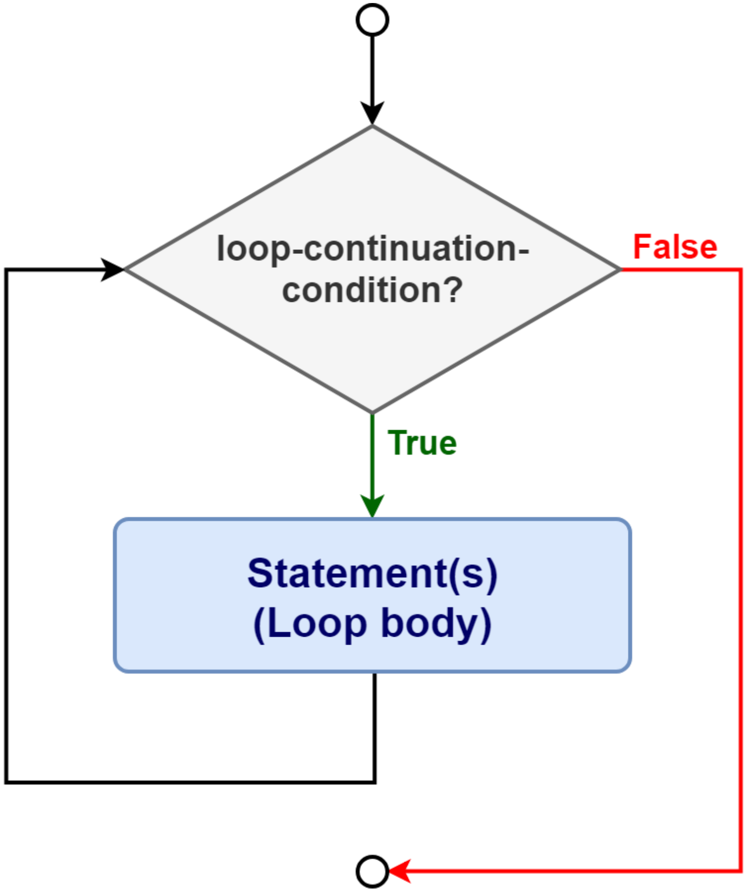
- Trace while Loop
- Infinite Loop
- Program 1: Subtraction Quiz
- Program 2: Guessing Game
- Loop Design Strategies
- Program 3: Multiple Subtraction Quiz
- Program 4: Advanced Multiple Subtraction Quiz
- Program 5: Sentinel Value
- Check Point #1 - #4

The while Loop

- A while loop executes statements repeatedly as long as a condition remains true.
- The syntax for the while loop is:

```
while loop-continuation-condition:
    # Loop body
    Statement(s)
```

  - The "loop body" is the part that contains the repeated statements.
  - A one-time execution of the loop body is called an iteration.
    - an iteration of the loop

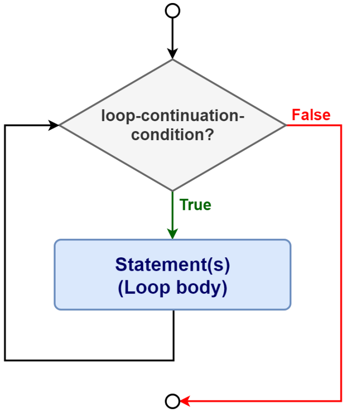


The while Loop

- The syntax for the while loop is:

```
while loop-continuation-condition:
    # Loop body
    Statement(s)
```

- Each loop contains a loop-continuation-condition.
- This is a Boolean expression that controls the execution of the loop body.
- This expression is evaluated at **each iteration**.
    - If the result is True, the loop body is executed
    - If it is False, the entire loop will terminate
        - Program control goes to the next statement after the loop.



The while Loop

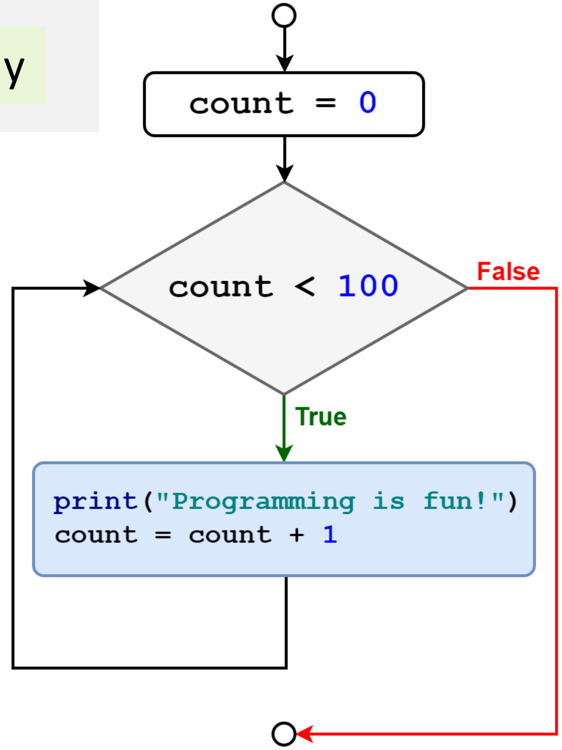
- The loop that displays Programming is fun! 100 times is an example of a while loop.

```
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

loop-continuation-condition

loop body

- The continuation condition is count < 100
  - If True, the loop continues.
  - If False, the loop will terminate.
- This type of loop is called a counter-controlled loop.



Trace while Loop

count → 0

```python
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

Initialize count

Trace while Loop

count → **0**

```python
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

(count < 2) is True

Trace while Loop

count → **0**

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

Print Programming is fun!

```
Programming is fun!
```

Trace while Loop

count → **1**

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

Increase count by **1**
count is **1** now

```
Programming is fun!
```

Trace while Loop

count → **1**

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

(count < 2) is still True since count is 1

```
Programming is fun!
```

Trace while Loop

count → 1

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

Print Programming is fun!

```
Programming is fun!
Programming is fun!
```

Trace while Loop

count → **2**

```python
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

Increase count by **1**
count is **2** now

```
Programming is fun!
Programming is fun!
```

Trace while Loop

count → 2

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

(count < 2) is False
since count is 2 now

```
Programming is fun!
Programming is fun!
```

Trace while Loop

count → 2

```
count = 0
while count < 2:
    print("Programming is fun!")
    count = count + 1
print("Done")
```

The loop exits. Execute the next statement after the loop.

Print Done

```
Programming is fun!
Programming is fun!
Done
```

The while Loop

- Here is another example illustrating how a loop works:

```python
sum = 0
i = 1
while i < 10:
    sum = sum + i
    i = i + 1
print("sum is", sum) # sum is 45
```

- Details:
  - if i < 10 is True, the program adds i to sum.
  - The variable i is initially set to **1**.
    - Then it is incremented to 2, 3, and so on, up to 10.
  - When i is 10, i < 10 is False, and the loop exits.
  - So the sum is 1 + 2 + 3 + … + 9 = **45**.

Infinite Loop

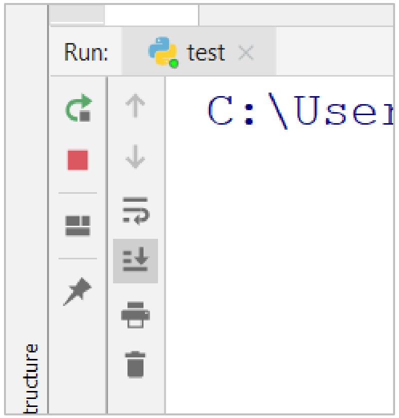
- Suppose the loop is mistakenly written as follows:

```
sum = 0
i = 1
while i < 10:
    sum = sum + i
i = i + 1
print("sum is", sum) # sum is 45
```

- Details:
  - Note that the entire loop body must be indented inside the loop.
  - Here the statement `i = i + 1` is not in the loop body.
  - This loop is infinite, because i is always **1** and `i < 10` will always be True.

Note

- Make sure that the loop-continuation-condition eventually becomes False.
  - So that the loop will terminate.
- A common programming error involves infinite loops.
  - The loop runs forever.
- If your program takes an unusual long time to run and does not stop, it may have an infinite loop.
- In PyCharm, click the small 🟥 in the bottom left corner.
  - This will stop the execution of the program.



```
C:\Use
```

Caution

- New programmers often make the mistake of executing a loop one extra time or one less time.
- This kind of mistake is commonly known as the off-by-one error.
- For example:

```
count = 0
while count <= 100:
    print("Programming is fun!")
    print(count)
    count = count + 1
```

- This displays "Programming is fun!" **101** times.
- Why?
  - count starts at **0**, which means it should go until count < 100.
  - If you want to iterate until count <= 100, then start count at **1**.

Subtraction Quiz

Program 1

Remember we wrote a program, in Chapter 4 (Program 3), to generate two numbers randomly and then ask the user for the answer of number1 - number2. Now, we rewrite this program to let the user repeatedly enter a new answer until it is correct.

```
What is 4 - 3? 4  <Enter>
Wrong answer. Try again. What is 4 - 3? 5  <Enter>
Wrong answer. Try again. What is 4 - 3? 1  <Enter>
You got it!
```

Subtraction Quiz
Phase 1: Problem-solving

Design your algorithm:

1. Generate two single-digit integers for number1 and number2.
   - Example: number1 = 4 and number2 = 3
2. If number1 < number2, swap number1 with number2.
   - Example: make number1 = 3 and number2 = 4
3. Ask the user to answer a question (answer)
   - Example: “What is 4 - 3 ?”
4. Make a while loop:
   - Condition of the loop is if the answer is wrong (number1 - number2 != answer)
   - If the answer is wrong, we should:
     - Ask the user to answer the question (answer) again.
5. Once the answer is finally correct:
   - Exit the loop.
   - Print "You got it!"

Subtraction Quiz
Phase 2: Implementation

LISTING 5.1 RepeatSubtractionQuiz.py

```python
import random

# 1. Generate two random single-digit integers
number1 = random.randint(0, 9)
number2 = random.randint(0, 9)

# 2. If number1 < number2, swap number1 with number2
if number1 < number2:
    number1, number2 = number2, number1

# 3. Prompt the student to answer What is number1 - number2?
answer = eval(input("What is " + str(number1) + " - "
                    + str(number2) + "? "))

# 4. Repeatedly ask the user the question until it is correct
while number1 - number2 != answer:
    answer = eval(input("Wrong answer. Try again. What is "
                        + str(number1) + " - " + str(number2) + "? "))

# 5. Print the answer is correct
print("You got it!")
```

Subtraction Quiz

Example Runs of The Program

```
What is 8 - 1? 7   <Enter>
You got it!
```

```
What is 6 - 3? 1   <Enter>
Wrong answer. Try again. What is 6 - 3? 2   <Enter>
Wrong answer. Try again. What is 6 - 3? 4   <Enter>
Wrong answer. Try again. What is 6 - 3? 3   <Enter>
You got it!
```

```
What is 7 - 4? 99   <Enter>
Wrong answer. Try again. What is 7 - 4? 5   <Enter>
Wrong answer. Try again. What is 7 - 4? 9   <Enter>
Wrong answer. Try again. What is 7 - 4? 8   <Enter>
Wrong answer. Try again. What is 7 - 4? 7   <Enter>
Wrong answer. Try again. What is 7 - 4? 3   <Enter>
You got it!
```

Guessing Game
Program 2

Write a program that randomly generates a number between **0** and **100**, inclusive. The program will repeatedly ask the user to guess the number until the user gets the number correct. At each wrong answer, the program tells the user if the guess is too low or too high.

```
Guess a magic number between 0 and 100
Enter your guess: 50  <Enter>
Your guess is too high
Enter your guess: 25  <Enter>
Your guess is too low
Enter your guess: 42  <Enter>
Your guess is too high
Enter your guess: 39  <Enter>
Yes, the number is 39
```

Guessing Game
Phase 1: Problem-solving

- This is the famous number guessing game.
- What is the normal first guess?
  - 50
- Why?
  - Because no matter the result (too high or too low), the number of possible answers left is divided in half!
    - If the guess is too high, you know the answer is in between 0 and 49.
    - If the guess is too low, you know the answer is in between 51 and 100.
    - So you can eliminate half of the numbers from consideration.

Guessing Game
Phase 1: Problem-solving

- So are we ready to code?
  - NO!
- We must THINK before coding.
- Think:
  - How would you solve the problem without a program?
  - You need a random number between 0 and 100.
  - You need to ask the user to enter a guess.
  - You need to compare the guess with the random number.

Guessing Game
Phase 1: Problem-solving

- Good idea to code incrementally when using loops
- Meaning:
    - Do not write the looping structure immediately.
    - First, try to just write the logic of the program, but without using loops.
        - So just write the code for one "loop", one iteration.
    - Then, write the code for the loop structure.
    - Think of the following code as an “initial draft”.

Guessing Game

Phase 2: Implementation (1st Draft)

LISTING 5.2 GuessNumberOneTime.py

```python
import random

# Generate a random number to be guessed
number = random.randint(0, 100)

print("Guess a magic number between 0 and 100")

# Prompt the user to guess the number
guess = eval(input("Enter your guess: "))

if guess == number:
    print("Yes, the number is " + str(number))
elif guess > number:
    print("Your guess is too high")
else:
    print("Your guess is too low")
```

Guessing Game
Phase 1: Problem-solving

- When this program runs, it prompts the user to enter a guess only once.

```
Guess a magic number between 0 and 100
Enter your guess: 50  <Enter>
Your guess is too high
```

- To let the user enter a guess repeatedly, you can change the code in lines 9–16 to create a loop, as follows:

```python
while True:
    # Prompt the user to guess the number
    guess = eval(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is " + str(number))
    elif guess > number:
        print("Your guess is too high")
    else:
        print("Your guess is too low")
```

Guessing Game

Phase 2: Implementation (2nd Draft)

GuessNumberInfiniteTime.py

```python
import random

# Generate a random number to be guessed
number = random.randint(0, 100)

print("Guess a magic number between 0 and 100")

while True:
    # Prompt the user to guess the number
    guess = eval(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is " + str(number))
    elif guess > number:
        print("Your guess is too high")
    else:
        print("Your guess is too low")
```

Guessing Game
Phase 1: Problem-solving

- This loop repeatedly prompts the user to enter a guess.

```
Guess a magic number between 0 and 100
Enter your guess: 25  <Enter>
Your guess is too low
Enter your guess: 39  <Enter>
Yes, the number is 39
Enter your guess: 42  <Enter>
Your guess is too high
Enter your guess: ...
```

- However, the loop doesn’t end even if the user entered the correct guess.
- This is because the condition of the loop is always True.
- So the loop still needs to terminate.
  - When the guess is finally correct, the loop should exit.

Guessing Game
Phase 1: Problem-solving

- So what is the loop condition?

```
while (guess != number)
```

  - So if the guess is not the same as the random number, continue the while loop.

- So, revise the loop as follows:

```
guess = -1 # Initial value that doesn't meet the loop condition
while guess != number:
    # Prompt the user to guess the number
    guess = eval(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is", number)
    elif guess > number:
        print("Your guess is too high")
    else:
        print("Your guess is too low")
```

Guessing Game

Phase 2: Implementation (Final)

LISTING 5.3 GuessNumber.py

```python
import random

# Generate a random number to be guessed
number = random.randint(0, 100)

print("Guess a magic number between 0 and 100")

guess = -1 # Initial value that doesn't meet the loop condition
while guess != number:
    # Prompt the user to guess the number
    guess = eval(input("Enter your guess: "))

    if guess == number:
        print("Yes, the number is", number)
    elif guess > number:
        print("Your guess is too high")
    else:
        print("Your guess is too low")
```

Guessing Game

Trace The Program Execution

```
Guess a magic number between 0 and 100
Enter your guess: 50  <Enter>
Your guess is too high
Enter your guess: 25  <Enter>
Your guess is too low
Enter your guess: 42  <Enter>
Your guess is too high
Enter your guess: 39  <Enter>
Yes, the number is 39
```

	line#	**number**	**guess**	**output**
	4	39		
	8		-1	
iteration 1	11		50	
	16			Your guess is too high
iteration 2	11		25	
	18			Your guess is too low
iteration 3	11		42	
	16			Your guess is too high
iteration 4	11		39	
	14			Yes, the number is 39

Guessing Game
Discussion

- The program generates the random number in line 4.
- Then, it prompts the user to enter a guess continuously in a loop (lines 9–18).
- For each guess, the program determines whether the user's number is correct, too high, or too low (lines 13–18).
- When the guess is correct, the program exits the loop (line 9).
- Note that guess is initialized to **-1**.
  - This is to avoid initializing it to a value between 0 and 100, because that could be the number to be guessed.

Loop Design Strategies

- Coding a correct loop is challenging for new programmers.
- Therefore, three steps are recommended:
  - Step 1: Identify the statements that need to be repeated.
  - Step 2: Wrap these statements in a loop (Infinite loop) like this:

```
while True:
    Statements
```

  - Step 3: Code the loop-continuation-condition and include appropriate statements to control the loop.

```
while loop-continuation-condition:
    Statements
    Additional statements for controlling the loop
```

Multiple Subtraction Quiz
Program 3

Write a program to randomly generate five subtraction questions and ask the user for the answer to each. Count how many the user got correct, and display the total time spent, by the user, answering the five questions.

```
What is 1 - 1? 0  <Enter>
You are correct!

What is 7 - 2? 5  <Enter>
You are correct!

What is 9 - 3? 4  <Enter>
Your answer is wrong.
9 - 3 is 6

What is 6 - 6? 0  <Enter>
You are correct!

What is 9 - 6? 2  <Enter>
Your answer is wrong.
9 - 6 is 3

Correct count is 3 out of 5
Test time is 10 seconds
```

Multiple Subtraction Quiz
Phase 1: Problem-solving

- Use the loop design strategy:
  - First, identify the statements that need to be repeated.
    - The statements for randomly generating two numbers.
    - Asking the user for the answer to the subtraction question.
    - Grading the question.
      - Comparing user answer to the real answer.
  - Second, wrap these statements inside a loop.
  - Finally, add a loop control variable and an appropriate loop-continuation-condition that will execute the loop five times.

Multiple Subtraction Quiz
Phase 2: Implementation

LISTING 5.4 SubtractionQuizLoop.py

```python
import random
import time

correctCount = 0 # Count the number of correct answers
count = 0 # Count the number of questions
NUMBER_OF_QUESTIONS = 5 # Constant

startTime = time.time() # Get start time

while count < NUMBER_OF_QUESTIONS:
    # 1. Generate two random single-digit integers
    number1 = random.randint(0, 9)
    number2 = random.randint(0, 9)

    # 2. If number1 < number2, swap number1 with number2
    if number1 < number2:
        number1, number2 = number2, number1

```

Multiple Subtraction Quiz
Phase 2: Implementation

LISTING 5.4 SubtractionQuizLoop.py

```
 # 3. Prompt the student to answer "what is number1 - number2?“
    answer = eval(input("What is " + str(number1) + " - " +
        str(number2) + "? "))

    # 5. Grade the answer and display the result
    if number1 - number2 == answer:
        print("You are correct!")
        correctCount += 1
    else:
        print("Your answer is wrong.\n", number1, "-",
            number2, "should be", (number1 - number2))

    # Increase the count
    count += 1

endTime = time.time() # Get end time
testTime = int(endTime - startTime) # Get test time
print("Correct count is", correctCount, "out of",
    NUMBER_OF_QUESTIONS, "\nTest time is", testTime, "seconds")
```

Multiple Subtraction Quiz
Discussion

- The program uses the control variable count to control the execution of the loop.
    - count is initially **0** (line 5)
    - count is increased by **1** in each iteration (line 32).
    - A subtraction question is displayed and processed in each iteration.
- The program obtains the time before the test starts (line 8) and the time after the test ends (line 34).
    - Then, it computes the test time in seconds (line 35).
- The program displays the correct count and test time after all the quizzes have been taken (lines 36–37).

Controlling a Loop with User Confirmation

- The preceding example (Program 3) executes the loop five times.
- If you want the user to decide whether to take another question, you can offer a user confirmation.
- The template of the program can be coded as follows:

```
continueLoop = 'Y'
while continueLoop == 'Y':
    # Execute the loop body once
    ...
    # Prompt the user for confirmation
    continueLoop = input("Enter Y to continue and N to quit: ")
```

Advanced Multiple Subtraction Quiz
Program 4

Rewrite Program 3 with user confirmation to let the user decide whether to advance to the next question.

```
What is 6 - 1? 5  <Enter>
You are correct!
Enter Y to continue and N to quit: Y  <Enter>

What is 8 - 0? 6  <Enter>
Your answer is wrong.
8 - 0 should be 8
Enter Y to continue and N to quit: Y  <Enter>

What is 8 - 3? 5  <Enter>
You are correct!
Enter Y to continue and N to quit: N  <Enter>

Correct count is 2 out of 3
Test time is 24 seconds
```

Advanced Multiple Subtraction Quiz
Phase 1: Problem-solving

- Use the template of controlling a loop with user confirmation:

```
continueLoop = 'Y'
while continueLoop == 'Y':
    # Execute the loop body once
    ...
    # Prompt the user for confirmation
    continueLoop = input("Enter Y to continue and N to quit: ")
```

- So we have to modify the loop condition as shown in the previous template.
    - Remove unnecessary variables or constants that the new loop condition doesn't use, such as NUMBER_OF_QUESTIONS.
    - After removing them, modify statements, such as print statements, that use the removed variables or contacts.

Advanced Multiple Subtraction Quiz
Phase 2: Implementation

SubtractionQuizLoopUserConfirmation.py

```python
import random
import time

correctCount = 0 # Count the number of correct answers
count = 0 # Count the number of questions

startTime = time.time() # Get start time

continueLoop = 'Y' # User confirmation flag
while continueLoop == 'Y':
    # 1. Generate two random single-digit integers
    number1 = random.randint(0, 9)
    number2 = random.randint(0, 9)

    # 2. If number1 < number2, swap number1 with number2
    if number1 < number2:
        number1, number2 = number2, number1

    # 3. Prompt the student to answer "what is number1 - number2?“
    answer = eval(input("What is " + str(number1) + " - " +
        str(number2) + "? "))
```

Advanced Multiple Subtraction Quiz

Phase 2: Implementation

SubtractionQuizLoopUserConfirmation.py

```python

    # 5. Grade the answer and display the result
    if number1 - number2 == answer:
        print("You are correct!")
        correctCount += 1
    else:
        print("Your answer is wrong.\n", number1, "-",
            number2, "should be", (number1 - number2))

    # Increase the count
    count += 1

    # Prompt the user for confirmation
    continueLoop = input("Enter Y to continue and N to quit: ")
    print() # To insert a new line

endTime = time.time() # Get end time
testTime = int(endTime - startTime) # Get test time
print("Correct count is", correctCount, "out of",
    count, "\nTest time is", testTime, "seconds")
```

Controlling a Loop with a Sentinel Value

- Often the number of times a loop is executed is not predetermined.
- Another common technique for controlling a loop is by choosing a special value when reading and processing user input.
- This special input value is known as a sentinel value.
- The sentinel value signifies the end of the input.
- A loop that uses a sentinel value in this way is called a sentinel-controlled loop.
- Example:
  - Ask the user to keep inputting as many integer values as they want.
  - Tell them that the loop will stop once the value **-1** is entered.
  - Therefore, **-1** would be the sentinel value.

Sentinel Value

Program 5

Write a program that will sum up all user inputted values. The user can keep inputting values for as long as the user wishes. If the user enters **"0"**, this means the end of the input. Your program should display the sum to the user.

```
Enter an integer (the input ends if it is 0): 2  <Enter>
Enter an integer (the input ends if it is 0): 3  <Enter>
Enter an integer (the input ends if it is 0): 4  <Enter>
Enter an integer (the input ends if it is 0): 0  <Enter>
The sum is 9
```

Sentinel Value
Phase 1: Problem-solving

- Use the loop design strategy:
  - First, identify the statements that need to be repeated.
    - Ask the user for a value (data).
      - `data = eval(input("Enter an integer ..."))`
    - Add it to the variable sum.
      - `sum += data`
  - Second, wrap these statements inside a loop.
  - Finally, add a loop control variable and an appropriate loop-continuation-condition which will be **data != 0** (the sentinel value).

Sentinel Value

Phase 2: Implementation

LISTING 5.5 SentinelValue.py

```python
data = eval(input("Enter an integer (the input exits " +
    "if the input is 0): "))

# Keep reading data until the input is 0
sum = 0
while data != 0:
    sum += data

    data = eval(input("Enter an integer (the input exits " +
        "if the input is 0): "))

print("The sum is", sum)
```

```
Enter an integer (the input ends if it is 0): 10  <Enter>
Enter an integer (the input ends if it is 0): 5  <Enter>
Enter an integer (the input ends if it is 0): 0  <Enter>
The sum is 15
```

Sentinel Value

Trace The Program Execution

```
Enter an integer (the input ends if it is 0): 2  <Enter>
Enter an integer (the input ends if it is 0): 3  <Enter>
Enter an integer (the input ends if it is 0): 4  <Enter>
Enter an integer (the input ends if it is 0): 0  <Enter>
The sum is 9
```

	line#	data	sum	output
	1	2		
	5		0	
iteration 1	7		2	
	9	3		
iteration 2	7		5	
	9	4		
iteration 3	7		9	
	9	0		
	12			The sum is 9

Sentinel Value

Discussion

- If data is **not** **0**, it is added to the sum (line 7) and the next item of input data is read (lines 9–10).
- If data is **0**, the loop body is no longer executed, and the while loop terminates.
- The input value **0** is the sentinel value for this loop.
- Note that if the first input read is **0**, the loop body never executes, and the resulting sum is **0**.

```
Enter an integer (the input ends if it is 0): 0  <Enter>
The sum is 0
```

Caution

- Don’t use floating-point values for equality checking in a loop control.
- **Why?**
  - Since those values are approximated, they could lead to **imprecise** counter values.
- Consider the following code for computing 1 + 0.9 + 0.8 + ... + 0.1:

```
item = 1
sum = 0
while item != 0: # No guarantee item will be 0
    sum += item
    item -= 0.1
print(sum)
```

Caution

- Consider the following code for computing 1 + 0.9 + ... + 0.1:

```
item = 1
sum = 0
while item != 0:
    sum += item
    item -= 0.1
print(sum)
```

	item	sum
Iteration 1 →	0.9	1
Iteration 2 →	0.8	1.9
Iteration 3 →	0.7000000000000001	2.7
Iteration 4 →	0.6000000000000001	3.4000000000000004
Iteration 5 →	0.5000000000000001	4.0
Iteration 6 →	0.40000000000000013	4.5
Iteration 7 →	0.30000000000000016	4.9
Iteration 8 →	0.20000000000000015	5.2
Iteration 9 →	0.10000000000000014	5.4
Iteration 10 →	0.00000000000000001	5.500000000000001

- The variable item starts with **1** and is reduced by **0.1** every time the loop body is executed.
- The loop should terminate when item becomes **0**.
- However, there is no guarantee that item will be **exactly 0**, because the floating-point arithmetic is approximated.
    - **0.00000000000000001 != 0** → True
- This loop seems okay on the surface, but it is actually an infinite loop.

Check Point

#1

Analyze the following code. Is count < 100 always True, always False, or sometimes True or sometimes False at **Point A**, **Point B**, and **Point C**?

```
count = 0
while count < 100:
    # Point A
    print("Programming is fun!")
    count += 1
    # Point B

# Point C
```

- Answer:
  - **Point A**: always True
  - **Point B**: sometimes False (Only one time)
  - **Point C**: always False

Check Point

#2

What is wrong if **guess** is initialized to **0** in line 8 in the following code?

LISTING 5.3 GuessNumber.py

```
 1  import random
 2
 3  # Generate a random number to be guessed
 4  number = random.randint(0, 100)
 5
 6  print("Guess a magic number between 0 and 100")
 7
 8  guess = -1 # Initial value that doesn't meet the loop condition
 9  while guess != number:
10      # Prompt the user to guess the number
11      guess = eval(input("Enter your guess: "))
12
13      if guess == number:
14          print("Yes, the number is", number)
15      elif guess > number:
16          print("Your guess is too high")
17      else:
18          print("Your guess is too low")
```

- Answer: the randomly generated number (number) could be **0**. If this happened, the program will not execute the loop.
  - The probability of this to be happened is 0.99% (1/101)

Check Point

#3

How many times are the following loop bodies repeated? What is the printout of each loop?

```python
i = 1
while i < 10:
    if i % 2 == 0:
        print(i)

```

(a)

```python
i = 1
while i < 10:
    if i % 2 == 0:
        print(i)
        i += 1
```

(b)

```python
i = 1
while i < 10:
    if i % 2 == 0:
        print(i)
    i += 1
```

(c)

Infinite Number of Times

Infinite Number of Times

9 Times

Empty

Empty

```
2
4
6
8
```

Check Point

#4

Suppose the input is 2 3 4 5 0 (one number per line). What is the output of the following code?

```
number = eval(input("Enter an integer: "))
max = number

while number != 0:
    number = eval(input("Enter an integer: "))
    if number > max:
        max = number

print("max is", max)
print("number", number)
```

```
Enter an integer: 2   <Enter>
Enter an integer: 3   <Enter>
Enter an integer: 4   <Enter>
Enter an integer: 5   <Enter>
Enter an integer: 0   <Enter>
max is 5
number 0
```

5.3. The for Loop

- Trace for Loop
- The range Function
- range(a, b)
- range(a)
- range(a, b, k)
- Check Point #5 - #9

The for Loop

- Often you will use a while loop to iterate a certain number of times.
- A loop of this type is called a counter-controlled loop.
- In general, the loop can be written as follows:

```
i = initialValue # Initialize loop-control variable
while i < endValue:
    # Loop body
    ...
    i += 1 # Adjust loop-control variable
```

- If you want to iterate a specific number of times, it is better/easier to just use a **for loop**.

The for Loop

```
i = initialValue # Initialize loop-control variable
while i < endValue:
    # Loop body
    ...
    i += 1 # Adjust loop-control variable
```

- A for loop can be used to simplify the preceding loop:

```
for i in range(initialValue, endValue):
    # Loop body
```

- In general, the syntax of a for loop is:

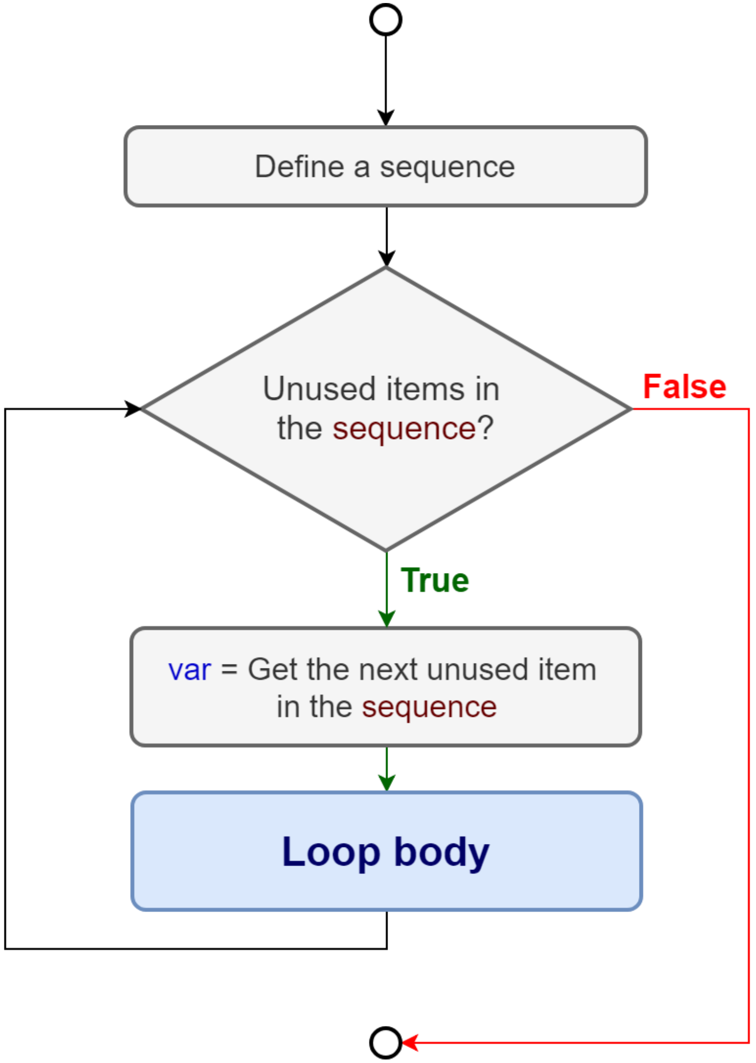
```
for var in sequence:
    # Loop body
```

The for Loop
Flowchart

- Flowchart for a generic for loop:

```
for var in sequence:
    # Loop body
```

- A sequence holds multiple items of data, stored one after the other.
- The variable var takes on each successive value in the sequence, the statements in the body of the loop are executed once for each value.

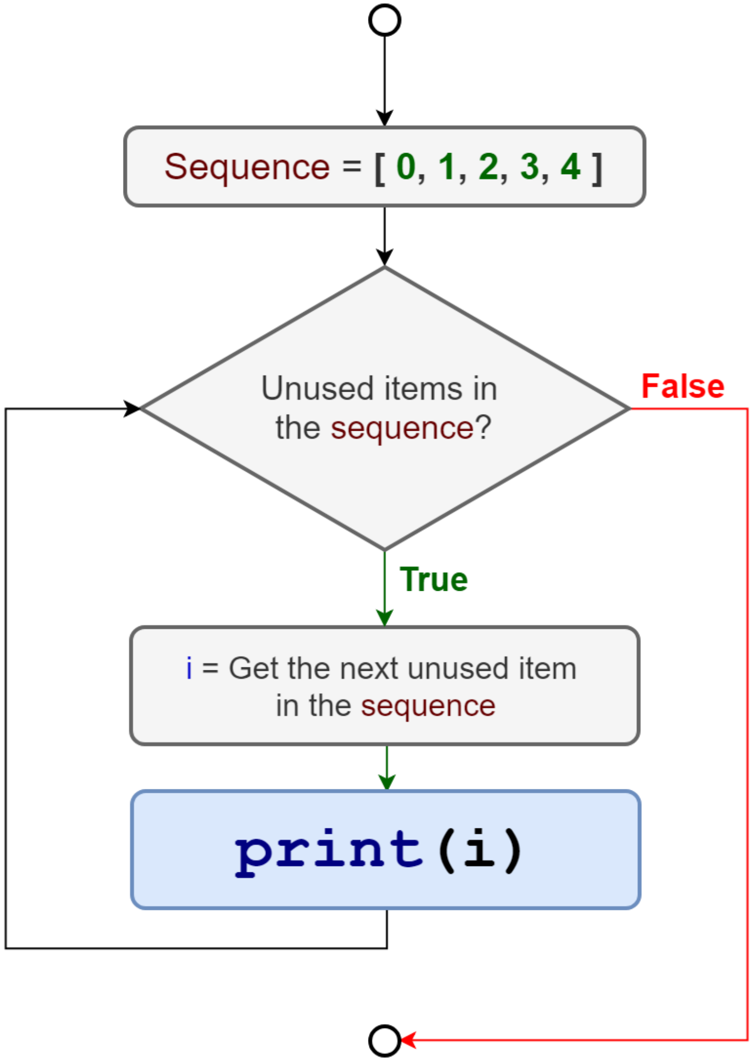


The for Loop
Example

- Example:

```python
for i in range(0,5):
    print(i)
```

```
0
1
2
3
4
```



Trace for Loop

```python
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print “Start ...”

```
Start ...
```

Trace for Loop

Range (0, 5) → **[0,1,2,3,4]**

i → **0**

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Loop 5 times.
Loop #1

```
Start ...
```

Trace for Loop

Range (0, 5) → **[0,1,2,3,4]**

i → **0**

```python
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print “i = 0”

```
Start ...
i = 0
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 1

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Loop #2

```
Start ...
i = 0
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 1

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print “i = 1”

```
Start ...
i = 0
i = 1
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 2

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Loop #3

```
Start ...
i = 0
i = 1
```

Trace for Loop

Range (0, 5) → **[0,1,2,3,4]**

↑

i → **2**

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print “i = 2”

```
Start ...
i = 0
i = 1
i = 2
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 3

```python
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Loop #4

```
Start ...
i = 0
i = 1
i = 2
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 3

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print “i = 3”

```
Start ...
i = 0
i = 1
i = 2
i = 3
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 4

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Loop #5

```
Start ...
i = 0
i = 1
i = 2
i = 3
```

Trace for Loop

Range (0, 5) → **[0,1,2,3,4]**

i → **4**

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print “i = 4”

```
Start ...
i = 0
i = 1
i = 2
i = 3
i = 4
```

Trace for Loop

Range (0, 5) → [0,1,2,3,4]

i → 4

```
print("Start ...")
for i in range(0,5):
    print("i =", i)
print("... End")
```

Print "... End"

```
Start ...
i = 0
i = 1
i = 2
i = 3
i = 4
... End
```

The range Function

- The range() function returns a sequence of integer numbers, starting from **0** by default, and increments by **1** (by default), and ends at a specified number.
- Syntax:

```
range(start, stop, step)
```

  - start: an integer number specifying at which position to start. Default is 0.
  - stop: an integer number specifying at which position to end.
  - step: an integer number specifying the incrementation. Default is 1

- It has three versions:
  - range(a)
  - range(a, b)
  - range(a, b, k)

range(a, b)

- The function range(a, b) returns the sequence of integers a, a + 1, ..., b - 2, and b - 1.
- For example:

```python
for v in range(4, 8):
    print("v =", v)
```

```
v = 4
v = 5
v = 6
v = 7
```

range(a)

- The function range(a) is the same as range(0, a).
- For example:

```
for v in range(6):
    print("v =", v)
```

```
v = 0
v = 1
v = 2
v = 3
v = 4
v = 5
```

range(a, b, k)

- k is used as step value in range(a, b, k).
  - The first number in the sequence is a.
  - Each successive number in the sequence will increase by the step value k.
  - b is the limit.
  - The last number in the sequence must be less than b.
- Example:

```
for v in range(3, 9, 2):
    print("v =", v)
```

```
v = 3
v = 5
v = 7
```

  - The step value in range (3, 9, 2) is **2**, and the limit is **9**. So, the sequence is 3, 5, and 7

range(a, b, k)
Count Backward

- The range(a, b, k) function can count backward if k is negative.
- In this case, the sequence is still a, a + k, a + 2k, and so on for a negative k.
- The last number in the sequence must be greater than b.
- Example:

```python
for v in range(5, 1, -1):
    print("v =", v)
```

```
v = 5
v = 4
v = 3
v = 2
```

Note

- The numbers in the range function must be integers.
- For example, range(1.5, 8.5), range(8.5), or range(1.5, 8.5, 1) would be wrong.
- Example:

```python
for v in range(6.5):
    print("v =", v)
```

```
 for v in range(6.5):
TypeError: 'float' object cannot be interpreted as
an integer
```

Note

`range(2)` → `[0, 1]`

`range(2,3)` → `[2]`

`range(2,-3)` → `[]`

`range(2,-3,-1)` → `[2, 1, 0, -1, -2]`

`range(2, 2)` → `[]`

`range(1, 2, 2)` → `[1]`

`range(2, 2, -1)` → `[]`

`range(5, 2, -1)` → `[5, 4, 3]`

Check Point

#5

Suppose the input is 2 3 4 5 0 (one number per line). What is the output of the following code?

```python
number = 0
sum = 0

for count in range(5):
    number = eval(input("Enter an integer: "))
    sum += number

print("sum is", sum)
print("count is", count)
```

```
Enter an integer: 2  <Enter>
Enter an integer: 3  <Enter>
Enter an integer: 4  <Enter>
Enter an integer: 5  <Enter>
Enter an integer: 0  <Enter>
sum is 14
count is 4
```

Check Point
#6

Can you convert any for loop to a while loop? List the advantages of using for loops.

- Answers:
  - Yes, we can convert any for loop to a while loop.
  - Advantages:
    - The number of repetitions is specified explicitly in advance.
    - When using a while loop, programmers often forget to adjust the control variable such as (i += 1). Using for loop can avoid this error.

Check Point

#7

Convert the following for loop statement to a while loop:

```python
sum = 0
for i in range(1001):
    sum = sum + i
print("sum =", sum)
```

```
sum = 500500
```

Solution:

```python
sum = 0
i = 0
while i < 1001:
    sum = sum + i
    i += 1
print("sum =", sum)
```

```
sum = 500500
```

Check Point

#8

Can you always convert any while loop into a for loop? Convert the following while loop into a for loop.

```
i = 1
sum = 0
while sum < 10000:
    sum = sum + i
    i += 1
print("sum =", sum)
```

```
sum = 10011
```

- Answers:
  - No, we cannot always convert any while loop into a for loop especially for the while loop that is not based on the counter variable (counter-controlled loop).

```
sum = 0
for i in range(1, 142):
    sum = sum + i
print("sum =", sum)
```

```
sum = 10011
```

Check Point

#9

How many times are the following loop bodies repeated? What is the printout of each loop?

```python
count = 0
while count < n:
    count += 1
```

(a)

n Times

```python
for count in range(n):
    print(count)

```

(b)

n Times

```python
count = 5
while count < n:
    count += 1
```

(c)

(n – count) Times

```python
count = 5
while count < n:
    count += 3
```

(d)

(The ceiling of (n - 5) / 3) Times

5.4. Nested Loops

- Trace Nested Loops
- Program 6: Multiplication Table
- Check Point #10 - #16

Nested Loops

- A loop can be nested inside another loop.
- Nested loops consist of an outer loop and one or more inner loops.
- Each time the outer loop is repeated, the inner loops are reentered and started anew.
- Example:

```
print("Start")
print()

for x in range(1, 4):
    print("----- x =", x, "-----")
    for y in range(4, 6):
        print("y =", y)
    print()

print("End")
```

Outer Loop

Inner Loop

```
Start

----- x = 1 -----
y = 4
y = 5

----- x = 2 -----
y = 4
y = 5

----- x = 3 -----
y = 4
y = 5

End
```

Trace Nested Loops

Draw a table and put each variable in a column.

Program Trace

i	j

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

↑

Loop 3 times, and the sequence is [1, 2, 3].
So, the first item is **1**. now i is **1**.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	

Trace Nested Loops

Range (1, 4) → **[1 , 2 , 3]**

j is 0 now

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

0 < 1 is True

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Print 0 and put a white space “ “ at the end.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
```

Program Trace

i	j
1	
	0

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Increment j by **1**. j is **1** now.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
```

Program Trace

i	j
1	
	0
	1

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

1 < 1 is False.
Exit from the current loop (inner loop).

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
```

Program Trace

i	j
1	
	0
	1

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Print a new line (\n)

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0
	1

```
0
```

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Update i to the next unused item in the sequence. Now i is **2**.

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
```

Program Trace

i	j
1	
	0
	1
2	

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

j is 0 now

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
```

Program Trace

i	j
1	
	0
	1
2	
	0

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

0 < 2 is True

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
```

Program Trace

i	j
1	
	0
	1
2	
	0

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Print 0 and put a white space “ “ at the end.

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0
	1
2	
	0

```
0
0
```

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Increment j by **1**. j is **1** now.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1

```
0
0
```

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

1 < 2 is True

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Print 1 and put a white space “ “ at the end.

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1

```
0
0 1
```

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Increment j by **1**. j is **2** now.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

2 < 2 is False.
Exit from the current loop (inner loop).

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2

```
0
0 1
```

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Print a new line (\n)

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2

Trace Nested Loops

Range (1, 4) → [1,2,3]

Update i to the next unused item in the sequence. Now i is **3**.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	

Trace Nested Loops

Range (1, 4) → [1,2,3]

j is 0 now

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0

Trace Nested Loops

Range (1, 4) → [1,2,3]

0 < 3 is True

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0

Trace Nested Loops

Range (1, 4) → [1,2,3]

Print 0 and put a white space " " at the end.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0

Trace Nested Loops

Range (1, 4) → [1,2,3]

Increment j by **1**. j is **1** now.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

1 < 3 is True

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1

Trace Nested Loops

Range (1, 4) → [1,2,3]

Print 1 and put a white space " " at the end.

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1

Trace Nested Loops

Range (1, 4) → [1,2,3]

Increment j by **1**. j is **2** now.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2

Trace Nested Loops

Range (1, 4) → [1,2,3]

2 < 3 is True

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2

Trace Nested Loops

Range (1, 4) → [1,2,3]

Print 2 and put a white space " " at the end.

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1 2
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

Increment j by **1**. j is **3** now.

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1 2
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2
	3

Trace Nested Loops

Range (1, 4) → [1 , 2 , 3]

3 < 3 is False.
Exit from the current loop (inner loop).

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1 2
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2
	3

Trace Nested Loops

Range (1, 4) → [1,2,3]

Print a new line (\n)

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1 2
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2
	3

Trace Nested Loops

Range (1, 4) → [1,2,3]

Is there any unused item in the sequence? No.
So, exit from the current loop (outer loop)

```
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1 2
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2
	3

Trace Nested Loops

Range (1, 4) → [1,2,3]

Print Done

```python
for i in range(1, 4):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
    print()
print("Done")
```

```
0
0 1
0 1 2
Done
```

Program Trace

i	j
1	
	0
	1
2	
	0
	1
	2
3	
	0
	1
	2
	3

Multiplication Table
Program 6

Write a program that uses nested for loops to print out the **1** through **9** multiplication table.

```
 Multiplication Table
  |   1   2   3   4   5   6   7   8   9
-----------------------------------------
1 |   1   2   3   4   5   6   7   8   9
2 |   2   4   6   8  10  12  14  16  18
3 |   3   6   9  12  15  18  21  24  27
4 |   4   8  12  16  20  24  28  32  36
5 |   5  10  15  20  25  30  35  40  45
6 |   6  12  18  24  30  36  42  48  54
7 |   7  14  21  28  35  42  49  56  63
8 |   8  16  24  32  40  48  56  64  72
9 |   9  18  27  36  45  54  63  72  81
```

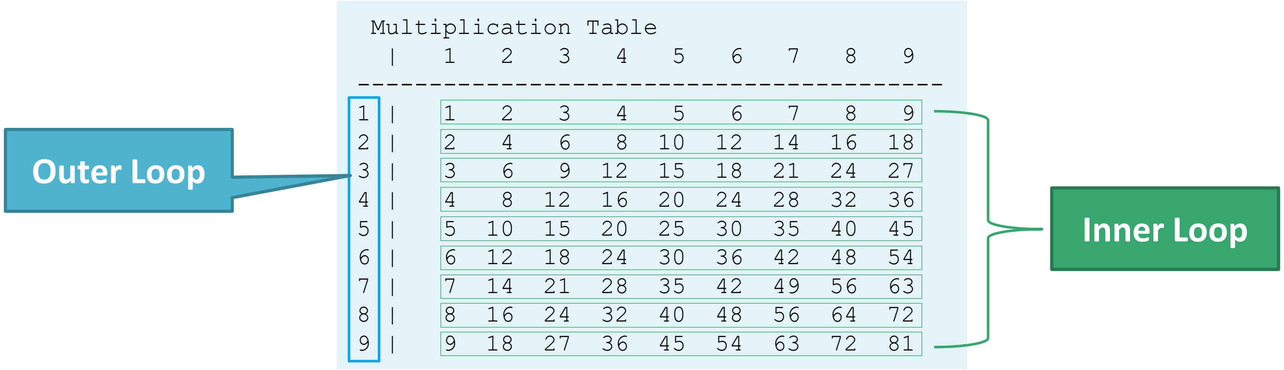
Multiplication Table
Phase 1: Problem-solving

- Examine the table:
  - We have nine rows of data.
  - We have nine columns of data.
- Look at the individual rows:
  - The first row contains the answer of 1x1, 1x2, 1x3…
  - The second row contains the answer of 2x1, 2x2, 2x3…
  - …
  - The ninth row contains the answer of 9x1, 9x2, 9x3…
- For each row heading (1, 2, 3, 4, 5, 6, 7, 8, 9):
  - We have that number multiplied by each of **1** through **9**.

Multiplication Table
Phase 1: Problem-solving

- Use two for loops:

1. The outer for loop will iterate over all rows
   - And it will start at i = 1
     - because the first row is labeled as 1.
   - And it will iterate until and including i = 9 (the last row).
2. Then, for EACH row, the inner for loop will calculate that row's answer of the row # times the values 1 through 9.



```
 Multiplication Table
  |  1    2    3    4    5    6    7    8    9
--------------------------------------------
1 |  1    2    3    4    5    6    7    8    9
2 |  2    4    6    8   10   12   14   16   18
3 |  3    6    9   12   15   18   21   24   27
4 |  4    8   12   16   20   24   28   32   36
5 |  5   10   15   20   25   30   35   40   45
6 |  6   12   18   24   30   36   42   48   54
7 |  7   14   21   28   35   42   49   56   63
8 |  8   16   24   32   40   48   56   64   72
9 |  9   18   27   36   45   54   63   72   81
```

Multiplication Table
Phase 2: Implementation

LISTING 5.6 MultiplicationTable.py

```
print("          Multiplication Table")
# Display the number title
print("  |", end = '')
for j in range(1, 10):
    print("  ", j, end = '')
print() # Jump to the new line
print("-----------------------------------------")

# Display table body
for i in range(1, 10):
    print(i, "|", end = '')
    for j in range(1, 10):
        # Display the product and align properly
        print(format(i * j, '4d'), end = '')
    print()# Jump to the new line
```

Multiplication Table

Discussion

- The program displays a title (line 1) on the first line in the output.
- The first for loop (lines 4–5) displays the numbers 1 through 9 on the second line.
- A line of dashes (-) is displayed on the third line (line 7).
- The next loop (lines 10–15) is a nested for loop with the control variable i in the outer loop and j in the inner loop.
- For each i, the product i * j is displayed on a line in the inner loop, with j being 1, 2, 3, . . ., 9.

Multiplication Table
Discussion

- To align the numbers properly, the program formats `i * j` using `format(i * j, "4d")` (line 14).
  - Recall that `"4d"` specifies a decimal integer format with width 4.
- Normally, the print function automatically jumps to the next line.
  - Invoking `print(item, end = '')` (lines 3, 5, 11, and 14) prints the item without advancing to the next line.
  - Note that the print function with the end argument was introduced in Chapter 3.

Note

- Be aware that a nested loop may take a long time to run.
- Consider the following loop nested in three levels:

```
for i in range(1000):
    for j in range(1000):
        for k in range(1000):
            Perform an action
```

  - The action is performed **1,000,000,000** times.
  - If it takes 1 millisecond to perform the action, the total time to run the loop would be more than **277 hours**.
- So be careful with many nested loops.

Check Point
#10

How many times is the print statement executed:

```
for i in range(10):
    for j in range(i):
        print(i * j)
```

- Solution:
  - The outer loop runs 10 times
    - From 0 to 9
  - For each iteration, the inner loop runs from 0 to i
    - First time i is 0, then i is 1, then 2, then 3, until i is 9
  - Answer: 1 + 2 + 3 + … + 9 = **45 times**

Check Point
#11

Show the output of the following programs. (Hint: Draw a table and list the variables in the columns to trace these programs.)

```python
for i in range(1, 5):
    j = 0
    while j < i:
        print(j, end = " ")
        j += 1
```

(a)

```
0 0 1 0 1 2 0 1 2 3
```

Program Trace

i	j
1	0
	1
2	0
	1
	2
3	0
	1
	2
	3
4	0
	1
	2
	3
	4

Check Point

#12

```python
i = 0
while i < 5:
    for j in range(i, 1, -1):
        print(j, end = " ")
    print("****")
    i += 1
```

(B)

```
****
****
2 ****
3 2 ****
4 3 2 ****
```

Program Trace

i	j
0	
1	
2	
	2
3	
	3
	2
4	
	4
	3
	2

Check Point
#13

```python
i = 5
while i >= 1:
    num = 1
    for j in range(1, i + 1):
        print(num, end = "xxx")
        num *= 2
    print()
    i -= 1
```

(c)

```
1xxx2xxx4xxx8xxx16xxx
1xxx2xxx4xxx8xxx
1xxx2xxx4xxx
1xxx2xxx
1xxx
```

Program Trace

i	num	j
5	1	
	2	1
	4	2
	8	3
	16	4
	32	5
4	1	
	2	1
	4	2
	8	3
	16	4
3	1	
	2	1
	4	2
	8	3
2	1	
	2	1
	4	2
1	1	
	2	1

Check Point
#14

```
i = 1
while i <= 5:
    num = 1
    for j in range(1, i + 1):
        print(num, end = "G")
        num += 2
    print()
    i += 1
```

(d)

```
1G
1G3G
1G3G5G
1G3G5G7G
1G3G5G7G9G
```

Program Trace

i	num	j
1	1	
	3	1
2	1	
	3	1
	5	2
3	1	
	3	1
	5	2
	7	3
4	1	
	3	1
	5	2
	7	3
	9	4
5	1	
	3	1
	5	2
	7	3
	9	4
	11	5

Check Point

#15

```
i = 5
while i >= 1:
    num = 1
    for j in range(1, i + 1):
        num *= 2
        print(num, end = "xxx")
    print()
    i -= 1
```

(e)

```
2xxx4xxx8xxx16xxx32xxx
2xxx4xxx8xxx16xxx
2xxx4xxx8xxx
2xxx4xxx
2xxx
```

Program Trace

i	num	j
5	1	
	2	1
	4	2
	8	3
	16	4
	32	5
4	1	
	2	1
	4	2
	8	3
	16	4
3	1	
	2	1
	4	2
	8	3
2	1	
	2	1
	4	2
1	1	
	2	1

Check Point
#16

```
i = 1
while i <= 5:
    num = 1
    for j in range(1, i + 1):
        num += 2
        print(num, end = "G")
    print()
    i += 1
```

(f)

```
3G
3G5G
3G5G7G
3G5G7G9G
3G5G7G9G11G
```

Program Trace

i	num	j
1	1	
	3	1
2	1	
	3	1
	5	2
3	1	
	3	1
	5	2
	7	3
4	1	
	3	1
	5	2
	7	3
	9	4
5	1	
	3	1
	5	2
	7	3
	9	4
	11	5

5.5. Minimizing Numerical Errors

Minimizing Numerical Errors

- Using floating-point numbers in the loop-continuation-condition may cause numeric errors.
- Numerical errors involving floating-point numbers are inevitable.
- This section provides an example showing you how to minimize such errors.

Minimizing Numerical Errors

Example

The following program sums a series that starts with **0.01** and ends with **1.0**. The numbers in the series will increment by **0.01**, as follows: 0.01 + 0.02 + 0.03 and so on.

LISTING 5.7 TestSum.py

```python
# Initialize sum
sum = 0

# Add 0.01, 0.02, ..., 0.99, 1 to sum
i = 0.01
while i <= 1.0:
    sum += i
    i = i + 0.01

# Display result
print("The sum is", sum)
```

```
The sum is 49.50000000000003
```

Minimizing Numerical Errors
Example

- The result displayed is 49.5, but the correct result is 50.5.
- What went wrong?
    - For each iteration in the loop, i is incremented by 0.01.
    - When the loop ends, the i value is slightly larger than **1** (not exactly **1**).
    - This causes the last i value not to be added into sum.
- The fundamental problem is that the floating-point numbers are represented by approximation.

Minimizing Numerical Errors
Example

- To fix the problem, use an integer count to ensure that all the numbers are added to sum.
- Here is the new loop:

TestSumFixWithWhileLoop.py

```python
# Initialize sum
sum = 0

# Add 0.01, 0.02, ..., 0.99, 1 to sum
i = 0.01
count = 0
while count < 100:
    sum += i
    i = i + 0.01
    count += 1 # Increase count

# Display result
print("The sum is", sum)
```

Minimizing Numerical Errors
Example

- Or, use a for loop as follows:

TestSumFixWithForLoop.py

```
# Initialize sum
sum = 0

# Add 0.01, 0.02, ..., 0.99, 1 to sum
i = 0.01
for count in range(100):
    sum += i
    i = i + 0.01

# Display result
print("The sum is", sum)
```

- After this loop, sum is 50.5.

```
The sum is 50.50000000000003
```

5.6. Case Studies

- Program 7: Finding the GCD
- Program 8: Predicting The Future Tuition

Finding the GCD

Program 7

Write a program to ask the user to enter two positive integers. You should then find the greatest common divisor (GCD) and print the result to the user.

```
Enter first integer: 125  <Enter>
Enter second integer: 2525  <Enter>
The greatest common divisor for 125 and 2525 is 25
```

Finding the GCD
Phase 1: Problem-solving

- Examples of Greatest Common Divisor (GCD):
  - The GCD of the two integers **4** and **2** is **2**
  - The GCD of the two integers **16** and **24** is **8**
  - The GCD of the two integers **25** and **60** is **5**
- So how do you calculate the GCD?
- Are you ready to code?
  - NO!
- Always, first think about the problem and understand the solution 100% before coding!
  - Thinking enables you to generate a logical solution for the problem without wondering how to write the code.
  - Once you have a logical solution, type the code to translate the solution into a program.

Finding the GCD
Phase 1: Problem-solving

- The GCD of the two integers number1 and number2:
    - You know that the number **1** is a common divisor.
        - because **1** divides into everything.
    - But is **1** the greatest common divisor?
    - So you can check the next values, one by one .
        - Check 2, 3, 4, 5, 6, …
        - Keep checking all the way up to the smaller of number1 or number2 .
    - Whenever you find a new common divisor, this becomes the new gcd.
    - After you check all the possibilities, the value in the variable gcd is the GCD of number1 and number2.

Finding the GCD

Phase 2: Implementation

LISTING 5.8 GreatestCommonDivisor.py

```python
# Prompt the user to enter two integers
n1 = eval(input("Enter first integer: "))
n2 = eval(input("Enter second integer: "))

gcd = 1 # Initial gcd is 1
k = 2 # Possible gcd
while k <= n1 and k <= n2:
    if n1 % k == 0 and n2 % k == 0:
        gcd = k # # Next possible gcd
    k += 1

print("The greatest common divisor for",
    n1, "and", n2, "is", gcd)
```

```
Enter first integer: 260  <Enter>
Enter second integer: 100  <Enter>
The greatest common divisor for 260 and 100 is 20
```

Predicting The Future Tuition

Program 8

A university charges **$10,000** per year for study (tuition). The cost of tuition increases **7%** every year. Write a program to determine how many years until the tuition will increase to **$20,000**.

```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

Information

Calculating Increasing/Decreasing By %

- How do you increase a number by x percent (x%)?
  - You can use the following formula:

    `Increase number by x% = number * ((100 + x) / 100)`
  - Example:
    - Suppose: number = **10000** and x= **7**
    - 7% increase for **10000** = **10000** * ((100 **+** **7**) / 100) = **10000** * **1.07** = 10700
- How do you decrease a number by x percent (x%)?
  - You can use the following formula:

    `Decrease number by x% = number * ((100 - x) / 100)`
  - Example:
    - Suppose: number = **10000** and x= **7**
    - 7% decrease for **10000** = **10000** * ((100 **-** **7**) / 100) = **10000** * **0.93** = 9300

Predicting The Future Tuition
Phase 1: Problem-solving

- Think:
  - How do we solve this on paper?
    - Cost of Year**0** = **$10,000** → Year**0** = 10,000
    - Cost of Year**1** = Year**0** * **1.07** → Year**1** = 10,000 * 1.07 = 10,700
    - Cost of Year**2** = Year**1** * **1.07** → Year**2** = 10,700 * 1.07 = 11,449
    - Cost of Year**3** = Year**2** * **1.07** → Year**3** = 11,449 * 1.07 = 12,250.43
    - …
    - Cost of Year**10** = Year**9** * **1.07** → Year**10** = 18384.59 * 1.07 = 19,671.51
    - Cost of Year**11** = Year**10** * **1.07** → Year**11** = 19671.51 * 1.07 = 21,048.51
  - So keep computing the tuition until it is at least **$20,000**.
  - Once you get to **$20,000**, print the number of years taken.

Predicting The Future Tuition
Phase 1: Problem-solving

- Think:
  - Now a closer look at some of the code:

```
year = 0 # Year 0
tuition = 10000

year += 1 # Year 1
tuition = tuition * 1.07

year += 1 # Year 2
tuition = tuition * 1.07

year += 1 # Year 3
tuition = tuition * 1.07
...
```

  - So we would keep doing this until tuition is greater than or equal to **$20,000**.
  - Then, at that point, we print the value in variable year.
  - How to do this? Use a while loop!

Predicting The Future Tuition

Phase 2: Implementation

LISTING 5.9 FutureTuition.py

```python
year = 0 # Year 0
tuition = 10000   # Year 1

while tuition < 20000:
    year += 1
    tuition = tuition * 1.07

print("Tuition will be doubled in", year, "years")
print("Tuition will be $" + format(tuition, ".2f"),
       "in", year, "years")
```

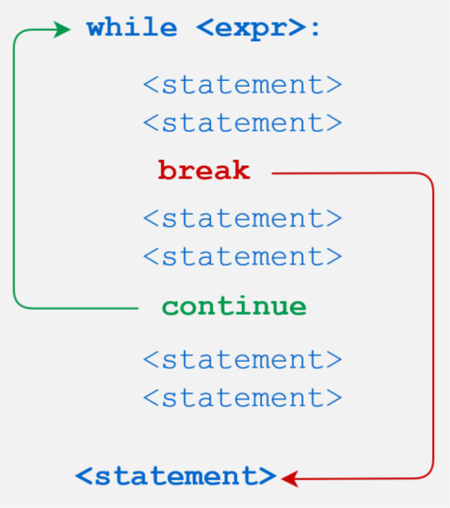
```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

5.7. Keywords break and continue

- break Keyword
- Trace break Statement
- continue Keyword
- Trace continue Statement
- When to Use break or continue?
- Check Point #17 - #22

Keywords break and continue

- The break and continue keywords provide additional controls to a loop.
- break keyword breaks out of **a loop**.
- continue keyword breaks out of **an iteration**.
- Benefits of using these keywords:
  - Can simplify programming in some cases.
- Negatives of using these keywords:
  - Overuse or improperly using them can cause problems and make programs difficult to read and debug.



```
while <expr>:
    <statement>
    <statement>
    break
    <statement>
    <statement>
    continue
    <statement>
    <statement>

<statement>
```

break Keyword

- You can use the keyword break in a loop to immediately terminate a loop.
- Example:

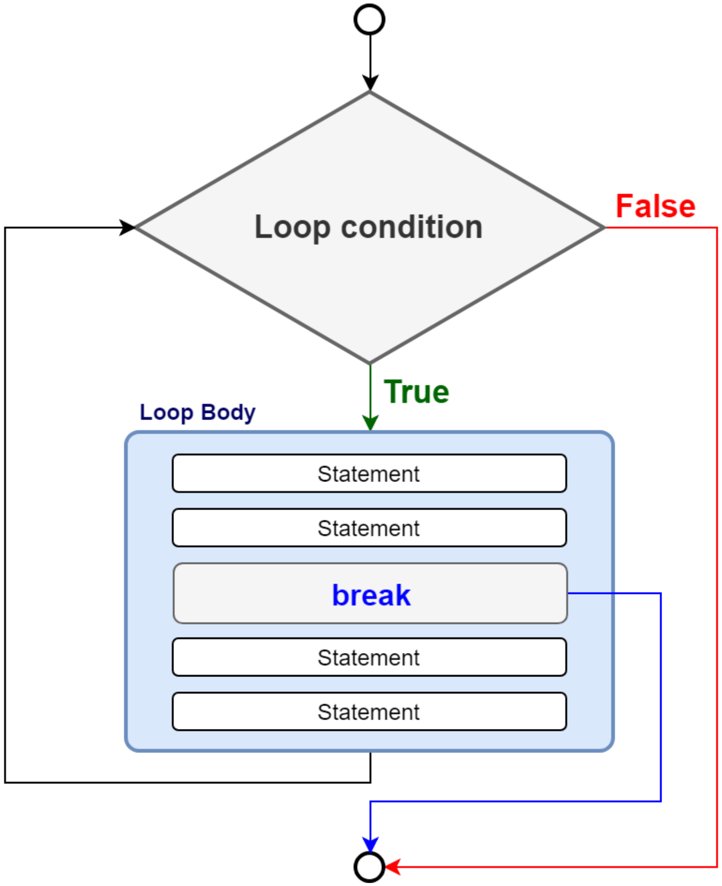
LISTING 5.11 TestBreak.py

```
sum = 0
number = 0

while number < 20:
    number += 1
    sum += number
    if sum >= 100:
        break

print("The number is", number)
print("The sum is", sum)
```

```
The number is 14
The sum is 105
```



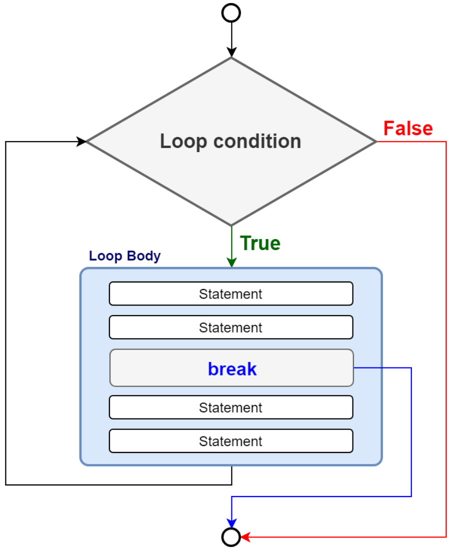
break Keyword

- Details:
  - The program adds integers from **1** to **20** in this order to sum until sum is greater than or equal to **100**.
  - **Without** lines 7–8, this program would calculate the sum of the numbers from **1** to **20**.
  - But with lines 7–8, the loop terminates when sum becomes greater than or equal to **100**.
  - **Without** lines 7–8, the output would be:

```
The number is 20
The sum is 210
```

LISTING 5.11 TestBreak.py

```
1   sum = 0
2   number = 0
3
4   while number < 20:
5       number += 1
6       sum += number
7       if sum >= 100:
8           break
9
10  print("The number is", number)
11  print("The sum is", sum)
```



Trace break Statement

sum → **0**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Initialize sum to **0**

Trace break Statement

sum → **0**

number → **0**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Initialize number to **0**

Trace break Statement

sum → **0**

number → **0**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

0 < 10 is True

Trace break Statement

sum → **0**

number → **0** **1**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Increment number by **1**

number = 0 + 1

Trace break Statement

sum → 0 1

number → 1

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Add number to sum

sum = 0 + 1

Trace break Statement

sum → **1**

number → **1**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

1 >= 5 is False

Trace break Statement

sum → **1**

number → **1**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

1 < 10 is True

Trace break Statement

sum → 1

number → 1 2

```python
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Increment number by **1**

number = 1 + 1

Trace break Statement

sum → 1 3

number → 2

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Add number to sum

sum = 1 + 2

Trace break Statement

sum → **3**

number → **2**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

3 >= 5 is False

Trace break Statement

sum → **3**

number → **2**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

2 < 10 is True

Trace break Statement

sum → 3

number → 2 3

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Increment number by **1**
number = 2 + 1

Trace break Statement

sum → 3 6   number → 3

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Add number to sum

sum = 3 + 3

Trace break Statement

sum → **6**

number → **3**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

6 >= 5 is True

Trace break Statement

sum → **6**

number → **3**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Exit the current loop

Trace break Statement

sum → **6**

number → **3**

```
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Print “The number is 3”

```
The number is 3
```

Trace break Statement

sum → **6**

number → **3**

```python
sum = 0
number = 0

while number < 10:
    number += 1
    sum += number
    if sum >= 5:
        break

print("The number is", number)
print("The sum is", sum)
```

Print “The sum is 6”

```
The number is 3
The sum is 6
```

continue Keyword

- You can use the continue keyword in a loop to end the **current** iteration, and program control goes to the end of the loop body.
- Example:

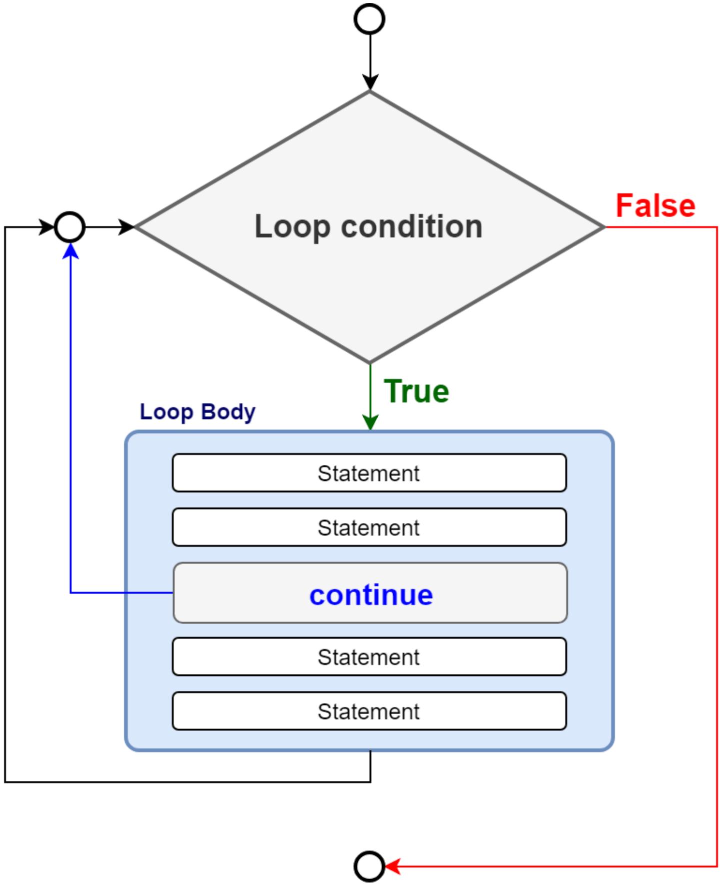
LISTING 5.12 TestContinue.py

```python
sum = 0
number = 0

while number < 20:
    number += 1
    if number == 10 or number == 11:
        continue
    sum += number

print("The sum is", sum)
```

```
The sum is 189
```



continue Keyword

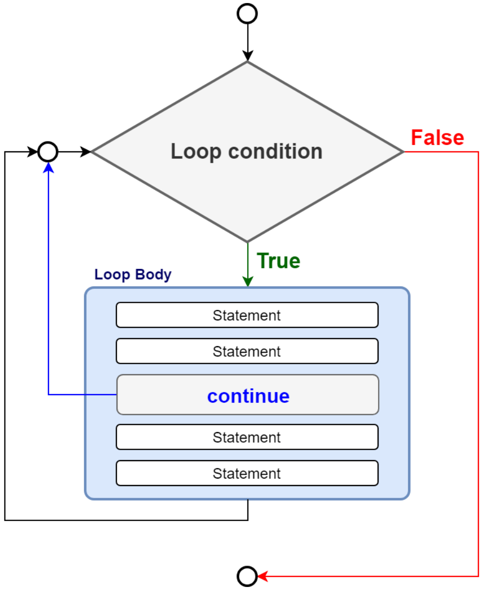
- Details:
  - The program adds all the integers from **1** to **20** except **10** and **11** to sum.
  - The continue statement is executed when number becomes **10** or **11**.
  - It ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, number is not added to sum when it is **10** or **11**.
  - **Without** lines 6 and 7, the output would be as follows:

```
The sum is 210
```

  - In this case, all the numbers are added to sum, even when number is **10** or **11**. Therefore, the result is **210**.

LISTING 5.12 TestContinue.py

```
1   sum = 0
2   number = 0
3
4   while number < 20:
5       number += 1
6       if number == 10 or number == 11:
7           continue
8       sum += number
9
10  print("The sum is", sum)
```



Trace continue Statement

sum → **0**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Initialize sum to **0**

Trace continue Statement

sum → **0**

number → **0**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Initialize number to **0**

Trace continue Statement

sum → **0**

number → **0**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

0 < 4 is True

Trace continue Statement

sum → **0**

number → **0** **1**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Increment number by **1**
number = 0 + 1

Trace continue Statement

sum → **0**

number → **1**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

(**1 == 1**) or (1 == 3) is

True

Trace continue Statement

sum → **0**

number → **1**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

End the current iteration

Trace continue Statement

sum → **0**

number → **1**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

1 < 4 is True

Trace continue Statement

sum → **0**

number → **1** **2**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Increment number by **1**
number = 1 + 1

Trace continue Statement

sum → **0**

number → **2**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

(2 == 1) or (2 == 3) is
False

Trace continue Statement

sum → 0 2

number → 2

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Add number to sum
sum = 0 + 2

Trace continue Statement

sum → **2**

number → **2**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

2 < 4 is True

Trace continue Statement

sum → 2

number → 2 3

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Increment number by **1**

number = 2 + 1

Trace continue Statement

sum → **2**

number → **3**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

(3 == 1) or (3 == 3) is
True

Trace continue Statement

sum → **2**

number → **3**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

End the current iteration

Trace continue Statement

sum → **2**

number → **3**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

3 < 4 is True

Trace continue Statement

sum → 2

number → 3 4

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Increment number by **1**

number = 3 + 1

Trace continue Statement

sum → **2**

number → **4**

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

(4 == 1) or (4 == 3) is
False

Trace continue Statement

sum → 2 | 6

number → 4

```
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Add number to sum

sum = 2 + 4

Trace continue Statement

sum → 6

number → 4

```python
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

4 < 4 is False

Trace continue Statement

sum → 6

number → 4

```python
sum = 0
number = 0

while number < 4:
    number += 1
    if number == 1 or number == 3:
        continue
    sum += number

print("The sum is", sum)
```

Print “The sum is 6”

```
The sum is 6
```

When to Use break or continue?

- You can always write a program without using break or continue in a loop.
- In general, it is appropriate to use break and continue if their use simplifies coding and makes programs easy to read.
- Suppose you need to write a program to find the smallest factor other than 1 for an integer n (assume n >= 2).
  - You can write a simple code using the break statement as follows:

```
n = eval(input("Enter an integer >= 2: "))
factor = 2
while factor <= n:
    if n % factor == 0:
        break
    factor += 1
print("The smallest factor other than 1 for", n, "is", factor)
```

Information
Factors of a Number

- The factors of a number are the numbers that divide evenly into the number.
- For example: the factors of the number **12** are the numbers **1**, **2**, **3**, **4**, **6** and **12**.
  - 12 % **1** = **0**
  - 12 % **2** = **0**
  - 12 % **3** = **0**
  - 12 % **4** = **0**
  - 12 % **5** = 2
  - 12 % **6** = **0**
  - 12 % **7** = 5
  - 12 % **8** = 4
  - 12 % **9** = 3
  - 12 % **10** = 2
  - 12 % **11** = 1
  - 12 % **12** = **0**
- Notice that the smallest factor is always 1 and the biggest factor is always the number itself.

When to Use break or continue?

- You may rewrite the code **without** using break as follows:

```
n = eval(input("Enter an integer >= 2: "))
found = False
factor = 2
while factor <= n and not found:
    if n % factor == 0:
        found = True
    else:
        factor += 1
print("The smallest factor other than 1 for", n, "is", factor)
```

- Obviously, the break statement makes the program simpler and easier to read in this example.
- However, you should use break and continue with caution.
  - Too many break and continue statements will produce a loop with many exit points and make the program difficult to read.

Check Point

#17

What is the keyword break for? What is the keyword continue for? Will the following program terminate? If so, give the output.

```
balance = 1000
while True:
    if balance < 9:
        break
    balance = balance - 9
print("Balance is", balance)
```

(a)

```
balance = 1000
while True:
    if balance < 9:
        continue
    balance = balance - 9
print("Balance is", balance)
```

(b)

- Answer:
  - The keyword break is used to exit the current loop.
  - The keyword continue causes the rest of the loop body to be skipped for the current iteration.
  - The program in (a) will terminate. The output is Balance is 1.
  - The while loop will **not** terminate in (b).

Check Point

#18

The for loop on the left is converted into the while loop on the right. What is wrong? Correct it.

```python
sum = 0
for i in range(4):
    if i % 3 == 0:
        continue
    sum += i
print(sum)
```

(a)

Converted

Wrong Conversion

```python
sum = 0
i = 0
while i < 4:
    if i % 3 == 0:
        continue
    sum += i
    i += 1
print(sum)
```

(b) Wrong

- Answer:
  - In (a), If the continue statement is executed inside the for loop, the rest of the iteration is skipped, then the loop control variable (i) is being updated to the next unused item in the sequence. **This code (a) is correct.**
  - In (b), If the continue statement is executed inside the while loop, the rest of the iteration is skipped, and the loop control variable (i) wouldn't get updated, so the loop condition will be always True. **This code (b) has an infinite loop.**

Check Point
#18

The for loop on the left is converted into the while loop on the right. What is wrong? Correct it.

- Here is the fix (b):

```
sum = 0
for i in range(4):
    if i % 3 == 0:
        continue
    sum += i
print(sum)
```

(a)

Converted

Correct Conversion

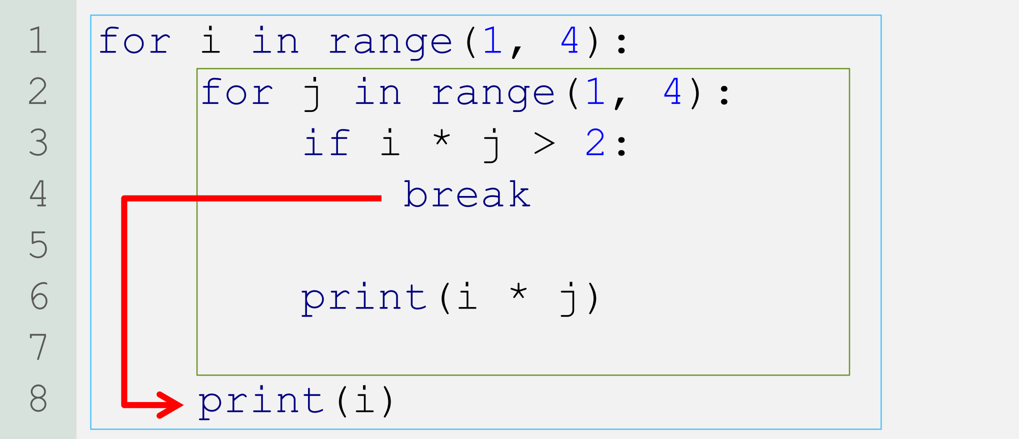
```
sum = 0
i = 0
while i < 4:
    if i % 3 == 0:
        i += 1
        continue
    sum += i
    i += 1
print(sum)
```

(b) Correct

Check Point

#19

After the **break** statement is executed in the following loop, which statement is executed? Show the output.



```
for i in range(1, 4):
    for j in range(1, 4):
        if i * j > 2:
            break

        print(i * j)

    print(i)
```

```
1
2
1
2
2
3
```

- Answer:
  - The **break** keyword immediately ends the innermost loop, which contains the **break**.
  - So, print(i) is the next statement that will be executed.

Check Point

#20

After the continue statement is executed in the following loop, which statement is executed? Show the output.

```
for i in range(1, 4):
    for j in range(1, 4):
        if i * j > 2:
            continue

        print(i * j)

    print(i)
```

```
1
2
1
2
2
3
```

- Answer:
  - The continue keyword ends **only** the current iteration.
  - If j is **not** the last item in the sequence, j is getting updated to the next unused item in the sequence, and if i * j > 2 is the next statement that will be executed.
  - If j **is** the last item in the sequence, print(i) is the next statement that will be executed.

Check Point
#21

Rewrite the following program **without** using break and continue statements.

LISTING 5.11 TestBreak.py

```python
sum = 0
number = 0

while number < 20:
    number += 1
    sum += number
    if sum >= 100:
        break

print("The number is", number)
print("The sum is", sum)
```

Solution →

TestBreakWithoutBreak.py

```python
sum = 0
number = 0
stop = False
while number < 20 and not stop:
    number += 1
    sum += number
    if sum >= 100:
        stop = True

print("The number is", number)
print("The sum is", sum)
```

Check Point

#22

Rewrite the following program **without** using break and continue statements.

LISTING 5.12 TestContinue.py

```python
sum = 0
number = 0

while number < 20:
    number += 1
    if number == 10 or number == 11:
        continue
    sum += number

print("The sum is", sum)
```

Solution →

TestContinueWithoutContinue.py

```python
sum = 0
number = 0

while number < 20:
    number += 1
    if number != 10 and number != 11:
        sum += number

print("The sum is", sum)
```

5.8. Case Study: Displaying Prime Numbers

- Program 9: Prime Number
- Coding with Loops

Prime Number
Program 9

Write a program to display the first 50 prime numbers in five lines (so 10 numbers per line).

- Note: any integer greater than **1** is prime if it can only be divided by 1 or itself.
- Example:
  - 2, 3, 5, and 7 are prime numbers
  - 4, 6, 8, and 9 are not prime numbers

```
The first 50 prime numbers are
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  127  131  137  139  149  151  157  163  167  173
  179  181  191  193  197  199  211  223  227  229
```

Prime Number
Phase 1: Problem-solving

- Think:
  - How can we solve this?
  - We need to check each integer greater than 1.
    - so start at 2, then 3, then 4, then 5, …
  - And for each of those integers, we need to check if it is prime.
  - If it is prime, we need to increase our count.
    - Because we found a new prime number.
  - And we also need to print it to the screen.
    - But we can only print 10 per line.
    - So we need to consider how many have been printed already.

Prime Number
Phase 1: Problem-solving

- Think:
  - So we need a loop!
  - How many times will we loop?
    - Many times.
    - Because we are checking each integer greater than 1 to determine if it is a prime number.
  - So will the loop go on for infinity?
    - No!
  - So for how long will the loop run?
    - Until we find and print **50** prime numbers!
    - Guess what: we now have our loop-continuation-condition!

Prime Number

Phase 2: Implementation (1^{st} Draft)

PrimeNumber.py

```
NUMBER_OF_PRIMES = 50  # Number of primes to display
NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
count = 0 # Count the number of prime numbers
number = 2 # A number to be tested for primeness
print("The first 50 prime numbers are")
# Repeatedly find prime numbers
while count < NUMBER_OF_PRIMES:
    # Assume the number is prime
    isPrime = True #Is the current number prime?

    # Test if number is prime
    # To do it later ...

    # If number is prime, display the prime number and increase the count
    if isPrime:
        count += 1 # Increase the count

        print(format(number, '5d'), end = '')
        if count % NUMBER_OF_PRIMES_PER_LINE == 0:
            # Display the number and advance to the new line
            print() # Jump to the new line

    # Check if the next number is prime
    number += 1
```

Prime Number

Phase 1: Problem-solving

- The output of the previous code:

```
The first 50 prime numbers are
    2    3    4    5    6    7    8    9   10   11
   12   13   14   15   16   17   18   19   20   21
   22   23   24   25   26   27   28   29   30   31
   32   33   34   35   36   37   38   39   40   41
   42   43   44   45   46   47   48   49   50   51
```

- So, we need now to filter the numbers and display only the prime numbers.
- This will be the next step.

Prime Number
Phase 1: Problem-solving

- Think:
  - Given a number, how can we determine if it is prime?
  - Check if it is divisible by 2, 3, 4, …, (number // 2)
    - If any of those values evenly divide number, then it is not prime.
  - So we use a for loop (from 2 until number // 2)
  - Example: consider the number 11.
    - Check from 2 to 5 (11//2 = 5)
      - 2 does not divide into 11
      - 3 does not divide into 11
      - 4 does not divide into 11
      - 5 does not divide into 11
      - Therefore, 11 is prime!

Prime Number
Phase 1: Problem-solving

- So, the code of the inner loop (filtering the non prime numbers) can be as the following:

```
# Assume the number is prime
isPrime = True #Is the current number prime?

# Test if number is prime
divisor = 2
while divisor <= number / 2:
    if number % divisor == 0:
        # If true, the number is not prime
        isPrime = False  # Set isPrime to false
        break  # Exit the for loop
    divisor += 1
```

Prime Number

Phase 2: Implementation (Final)

LISTING 5.13 PrimeNumber.py

```
NUMBER_OF_PRIMES = 50  # Number of primes to display
NUMBER_OF_PRIMES_PER_LINE = 10 # Display 10 per line
count = 0 # Count the number of prime numbers
number = 2 # A number to be tested for primeness

print("The first 50 prime numbers are")

# Repeatedly find prime numbers
while count < NUMBER_OF_PRIMES:
    # Assume the number is prime
    isPrime = True #Is the current number prime?

    # Test if number is prime
    divisor = 2
    while divisor <= number / 2:
        if number % divisor == 0:
            # If true, the number is not prime
            isPrime = False  # Set isPrime to false
            break  # Exit the for loop
        divisor += 1

```

Prime Number
Phase 2: Implementation (Final)

LISTING 5.13 PrimeNumber.py

```python
# If number is prime, display the prime number and increase the count
if isPrime:
    count += 1 # Increase the count

    print(format(number, '5d'), end = '')
    if count % NUMBER_OF_PRIMES_PER_LINE == 0:
        # Display the number and advance to the new line
        print() # Jump to the new line

# Check if the next number is prime
number += 1
```

Coding with Loops

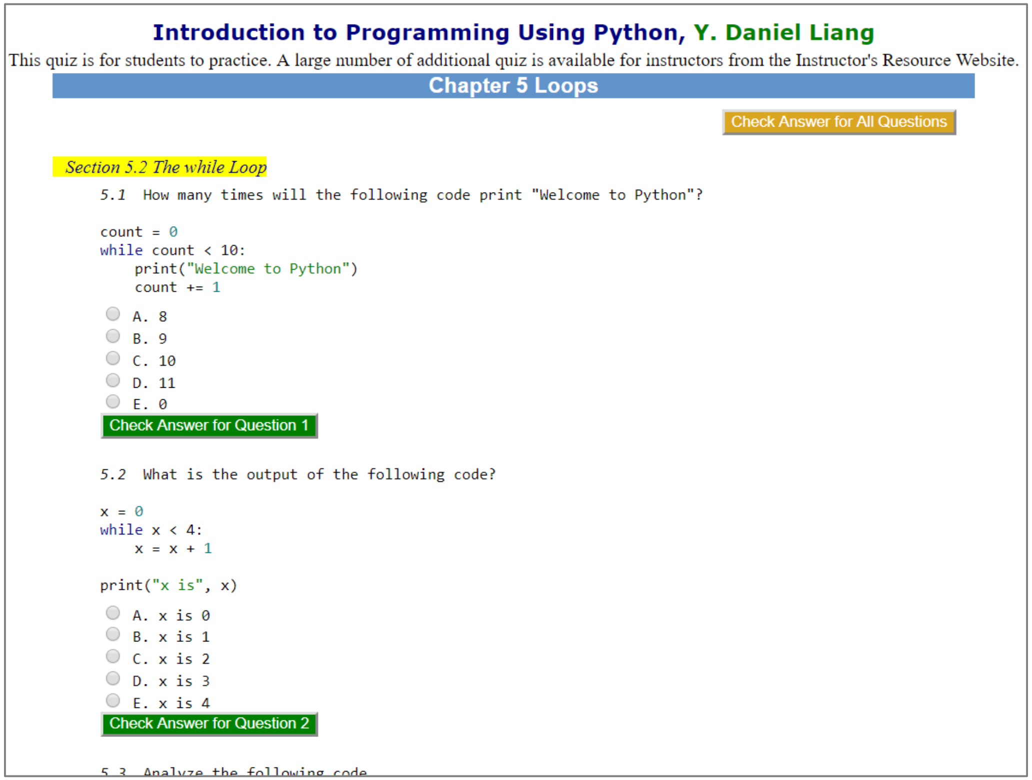
- This last example (Program 9) was complicated.
- If you understand it, congratulations!
- Question:
  - How can new programmers develop a solution similar to what we just did?
- Answer:
  - Break the problem into smaller sub-problems.
  - Develop solutions for each of those sub-problems.
- Then bring the smaller solutions together into one larger solution.

End

- Test Questions
- Programming Exercises

Test Questions

- Do the test questions for this chapter online at https://liveexample-ppe.pearsoncmg.com/selftest/selftestpy?chapter=5



Introduction to Programming Using Python, Y. Daniel Liang

This quiz is for students to practice. A large number of additional quiz is available for instructors from the Instructor's Resource Website.

Chapter 5 Loops

Section 5.2 The while Loop

5.1 How many times will the following code print "Welcome to Python"?

```
count = 0
while count < 10:
    print("Welcome to Python")
    count += 1
```

○ A. 8
○ B. 9
○ C. 10
○ D. 11
○ E. 0

5.2 What is the output of the following code?

```
x = 0
while x < 4:
    x = x + 1

print("x is", x)
```

○ A. x is 0
○ B. x is 1
○ C. x is 2
○ D. x is 3
○ E. x is 4

5.3 Analyze the following code

Programming Exercises

- Page 158 – 167:
  - 5.1 - 5.16
  - 5.18 - 5.22
  - 5.23 – 5.41
  - 5.43 – 5.46