Chapter 4: Loops 1 & 2

REVISED SPRING 2023 – A.M.TSAASAN W/ G. ERNSBERGER

Chapter Goals

- To implement while and for loops
- To hand-trace the execution of a program
- To become familiar with common loop algorithms
- To understand nested loops
- To implement programs that read and process data sets
- To use a computer for simulations

In this chapter, you will learn about loop statements in Python, as well as techniques for writing programs that simulate activities in the real world.

Contents

Decisions Review

```
given = 4
if given - 2 > 2:
      given = given + 1
else:
      given = given - 1
```

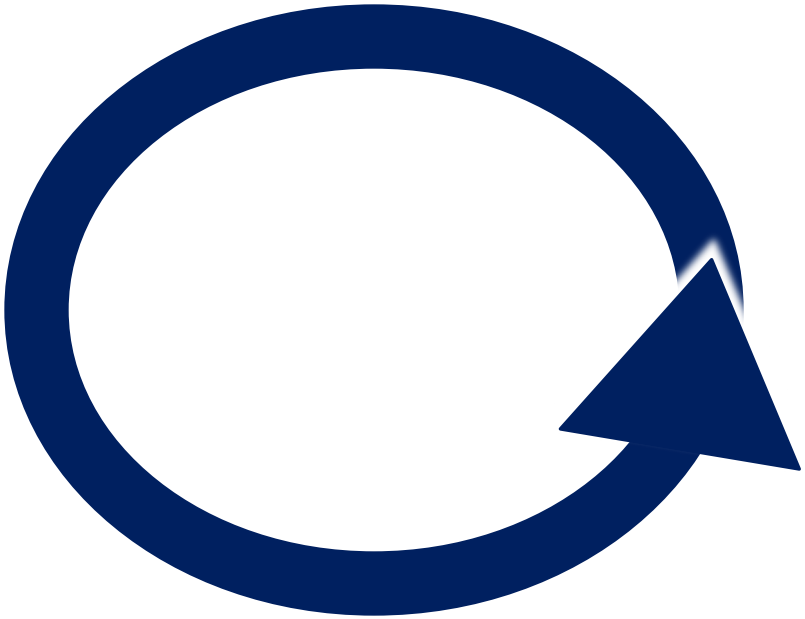
What is the final value of given?

A. 1
B. 2
C. 3
D. 4
E. 5

```
given = 5
if given - 2 > 2:
      given = given + 1
else:
      given = given - 1
```

What is the final value of given?

A. 2
B. 3
C. 4
D. 5
E. 6



Loops in computing

ALL LANGUAGES…

LOOPS MEAN THE SAME THING, DO THE SAME THING

What is a Loop?

- A ***loop*** is a portion of a program that **repeats** a statement or a group of statements.
- Why is repetition needed?
  - There are many situations in which the same statements need to be executed multiple times.
  - Example:
    - Asking the user to enter grades to compute the average
    - How many times would you need to write the statement that reads the input?

Example – try it ? Think nested!

- Sample screen output:

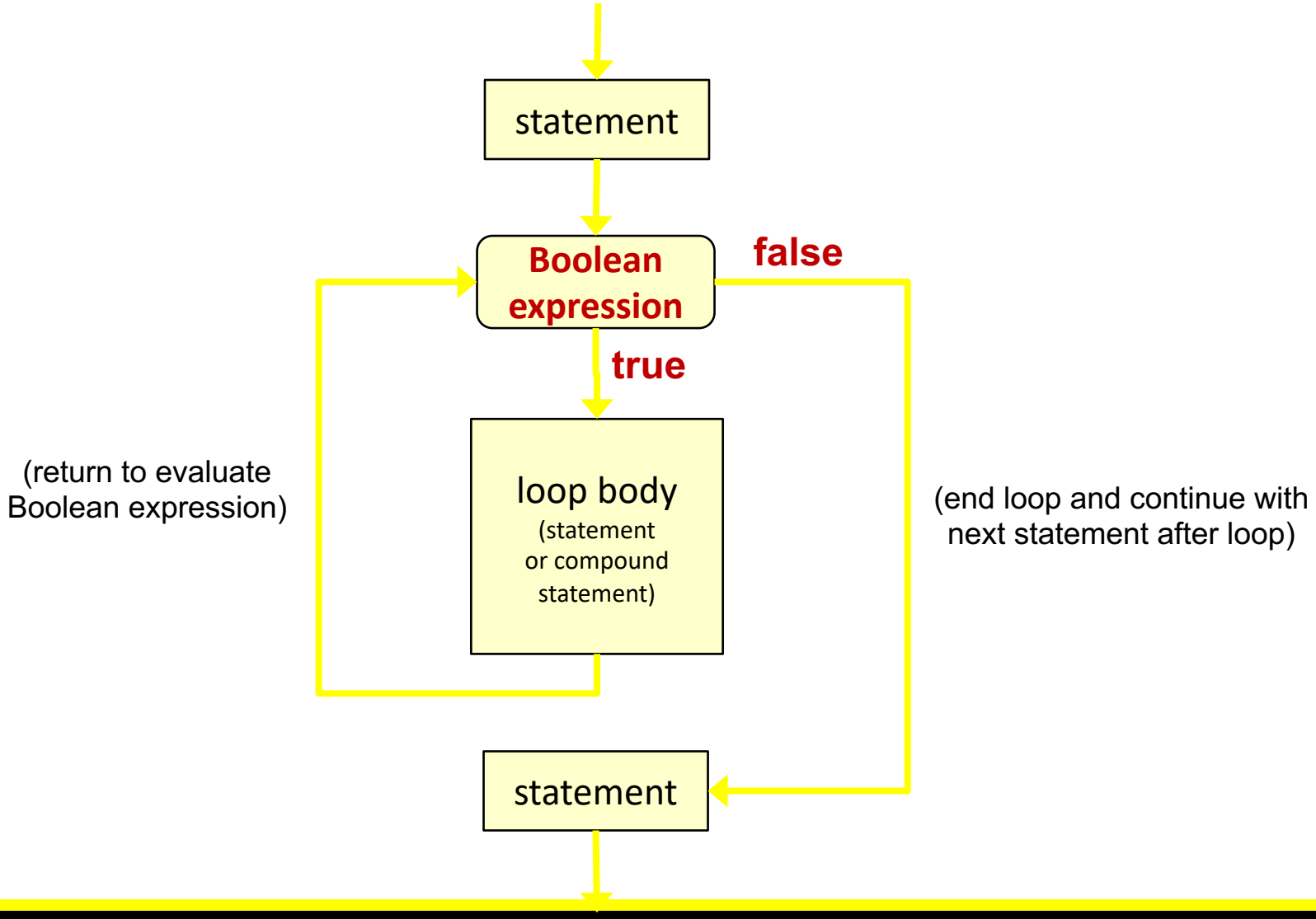
```
Enter all the scores to be averaged.
Enter a negative number after you are done.
90
70
80
70
70
80
90
90
80
-1

The average is 80.0
```

The **while** Loop

The **`while`** Loop

- A **`while`** statement **repeats** while a ***controlling boolean expression*** remains **`true`**.
- The ***loop body*** typically contains an action that ultimately causes the *controlling boolean expression* to become **`false`**.

The **`while`** Loop



Count-Controlled Loops

```
counter = 1                          # Initialize the counter
while counter <= 10 :                # Check the counter
  print(counter)
  counter = counter + 1              # Update the loop variable
```

Syntax of a **`while`** Loop

- Syntax:

```
while (Boolean_Expression):
    Statements
    ...
```

- Example:

```
count = 0

while ( count <= 20 ):
    print(count)
    count += 5
```

Output:

```
0
5
10
15
20
```

Syntax of a **`while`** Loop

- **Declaration and initialization of control variable**
  - Declare and initialize the variable that will control the number of times the loop will be executed.

```
count = 0

while ( count <= 20 ):
    print(count)
    count += 5
```

Syntax of a **`while`** Loop

- **Control statement**
  - This loop will continue to execute as long as the variable **`count`** is less than or equal to 20.

```
count = 0

while (count <= 20):
    print(count)
    count += 5
```

Syntax of a **`while`** Loop

- **Loop body**
  - The statements within the indentation will continue to execute until count is greater than 20.

```
count = 0

while (count <= 20):
    print(count)
    count += 5
```

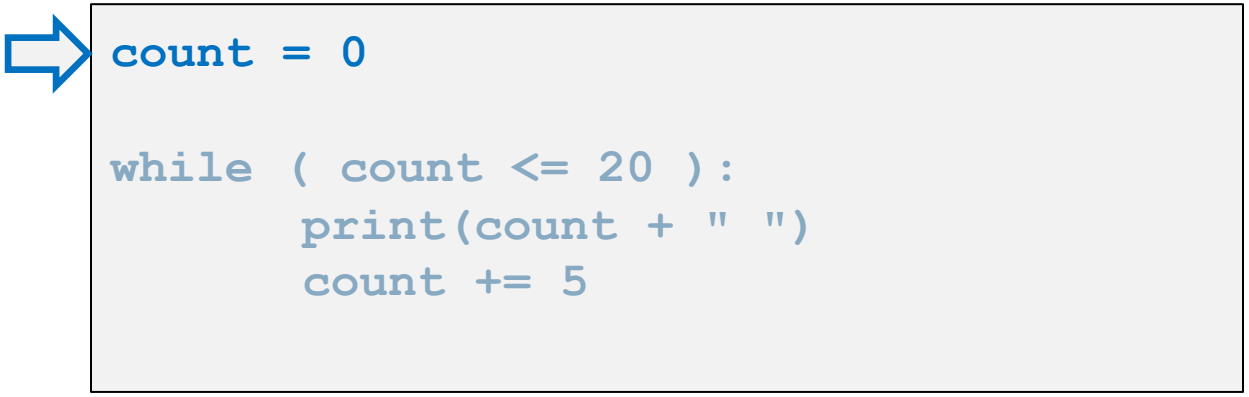
Syntax of a **`while`** Loop

- **Loop control variable**
  - Needs to be updated to eventually stop the loop
  - An ***infinite loop*** occurs if the control variable is not updated.

```
count = 0


while (count <= 20):
      print(count)
      count += 5
```

Execution of a **`while`** Loop

- Let's look at this loop step by step.
- Variable **`count`** with value 0 is stored in memory.



```
count = 0

while ( count <= 20 ):
        print(count + " ")
        count += 5
```

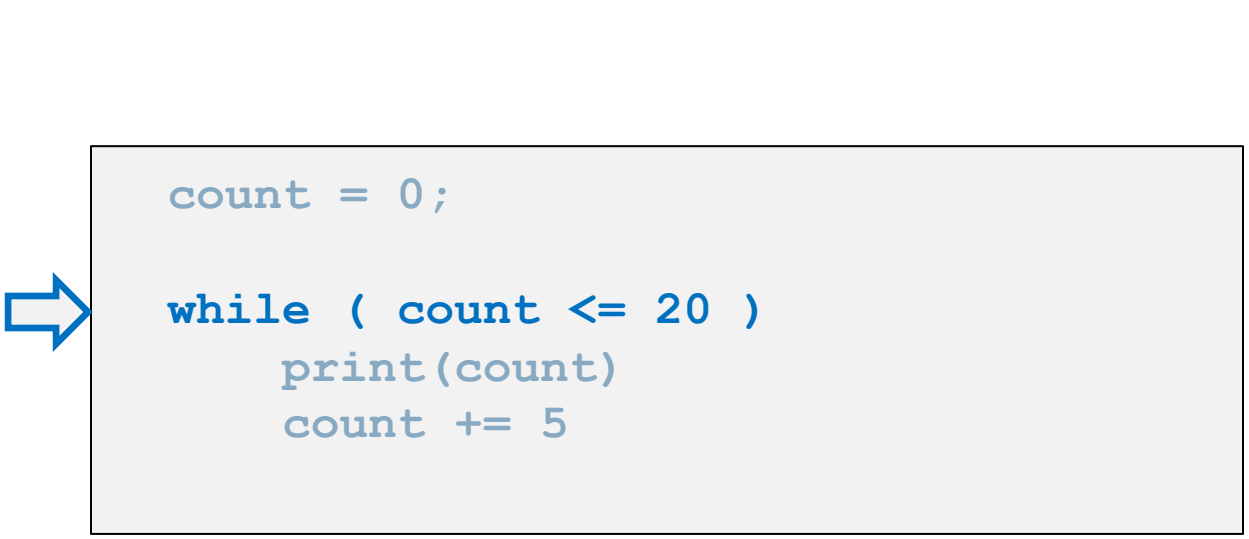
`count`

0

`Output`

Execution of a **`while`** Loop

- **Control statement** checks if **`count`** is ***less or equal*** to 20 → **`true`** → executes the body of the loop.



```
count = 0;

while ( count <= 20 )
    print(count)
    count += 5
```

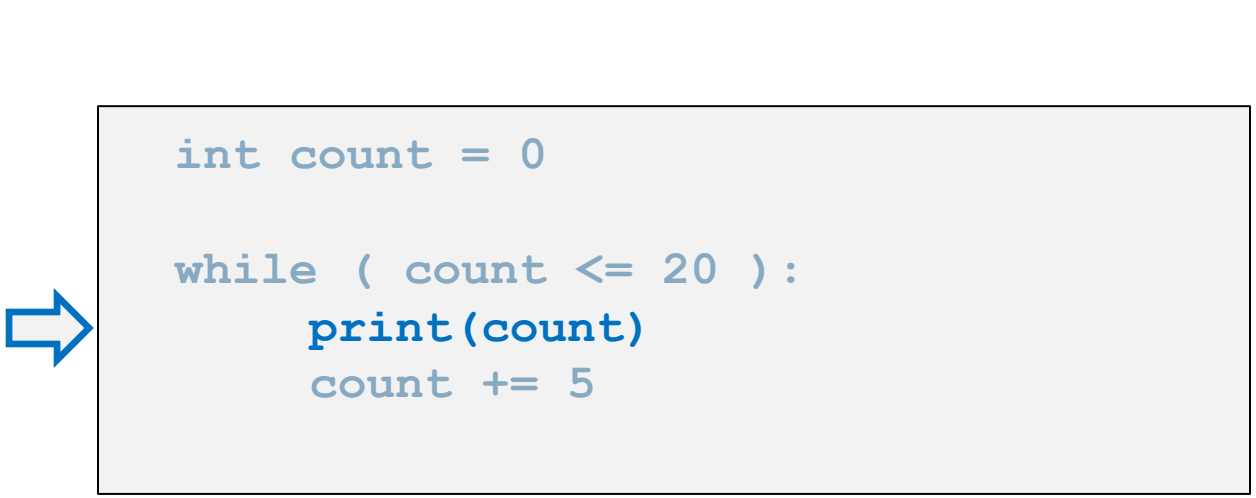
`count`

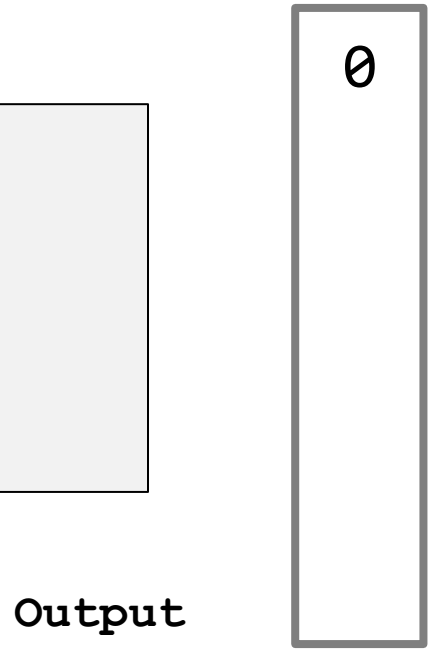
0

`Output`

Execution of a **`while`** Loop

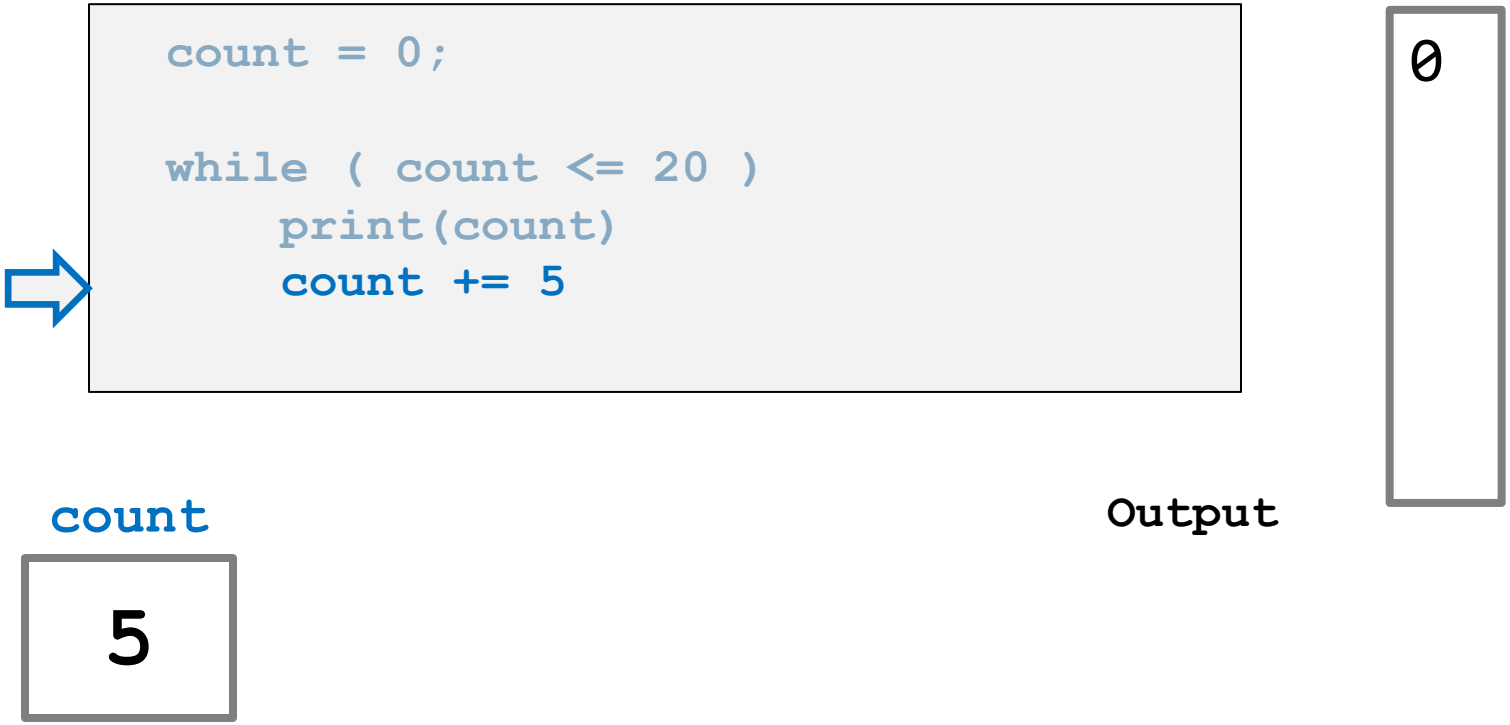
- First statement is executed and prints out 0.





Execution of a **`while`** Loop

- Second statement is executed and increases the value of the variable **`count`** by 5.



Execution of a **`while`** Loop

- **Control statement** checks if **`count`** is ***less or equal*** to 20 → **`true`** → executes the body of the loop.

```
count = 0

while ( count <= 20 )
      print(count)
      count += 5
```

`count`

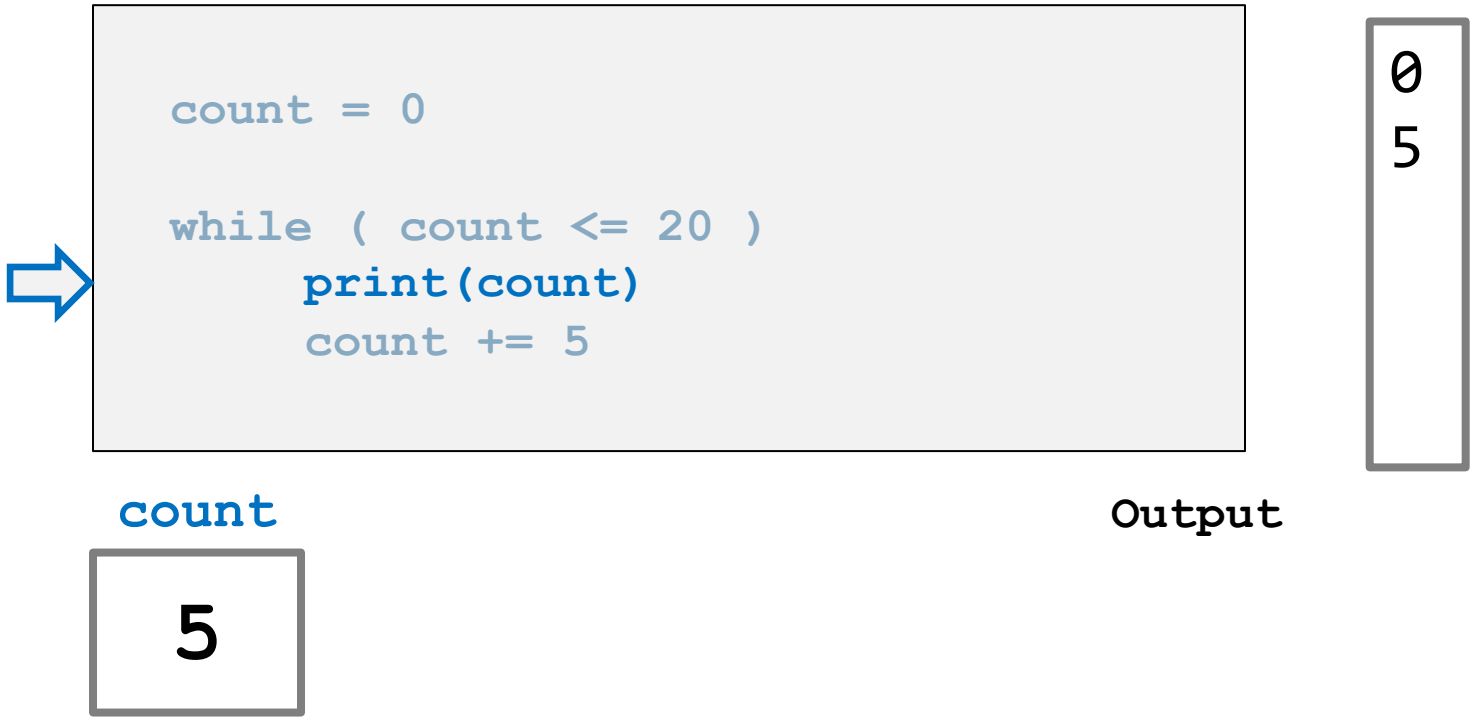
5

`Output`

0

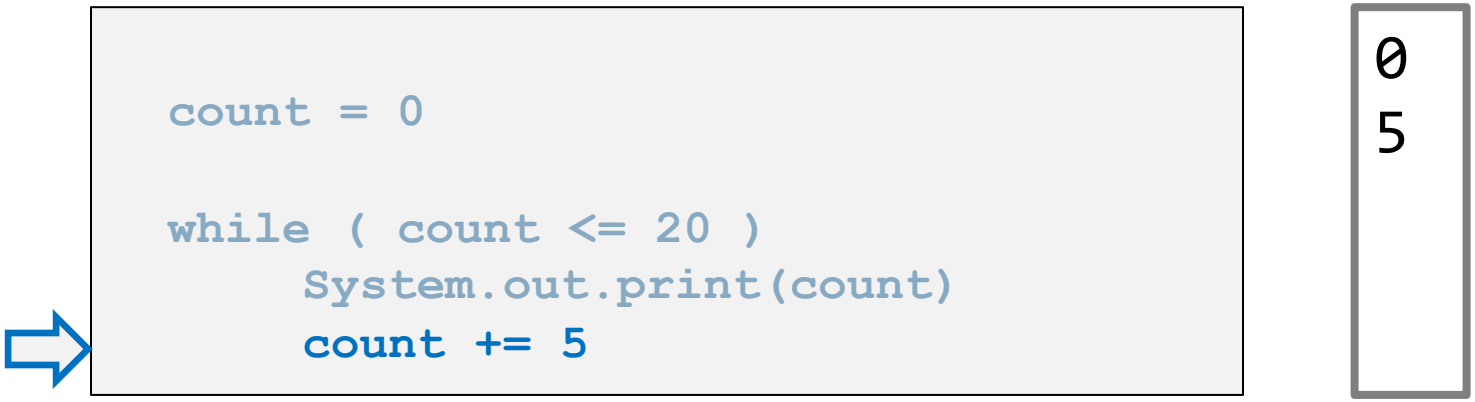
Execution of a **`while`** Loop

- First statement is executed and prints out 5.



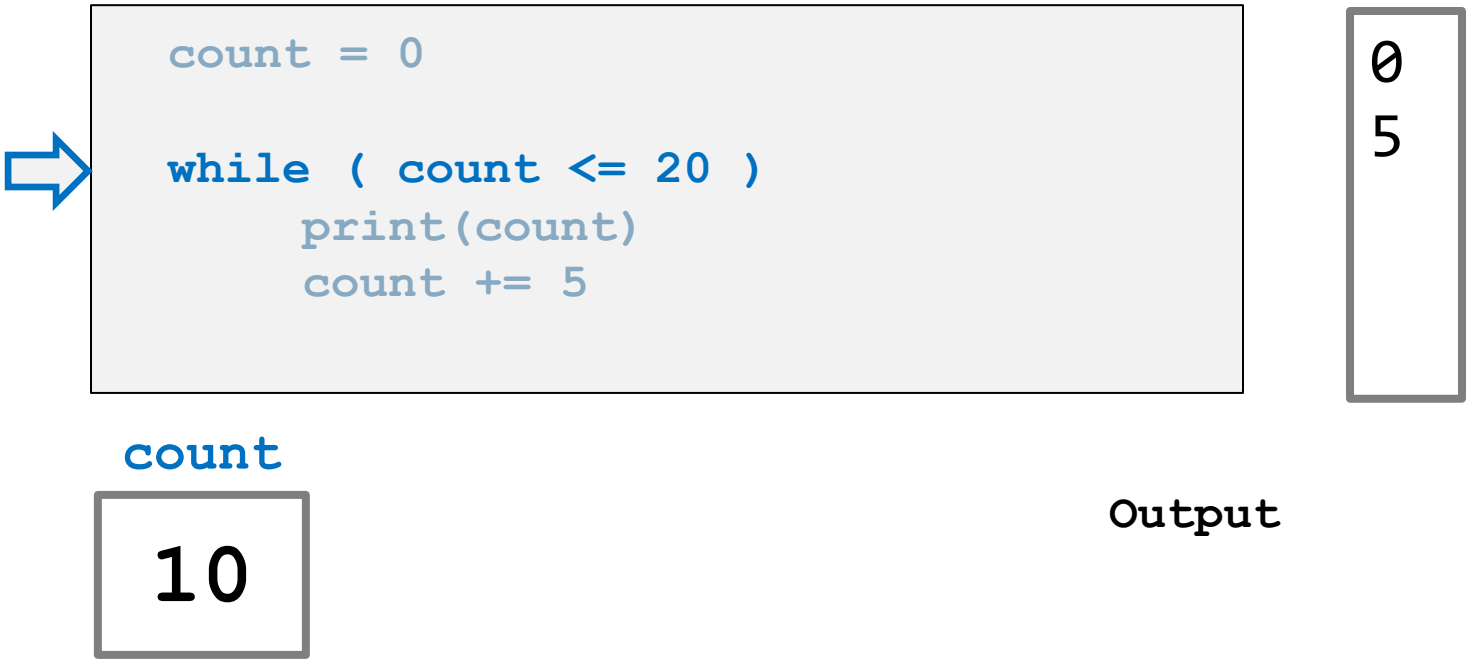
Execution of a **`while`** Loop

- Second statement is executed and increases the value of the variable **`count`** by 5.



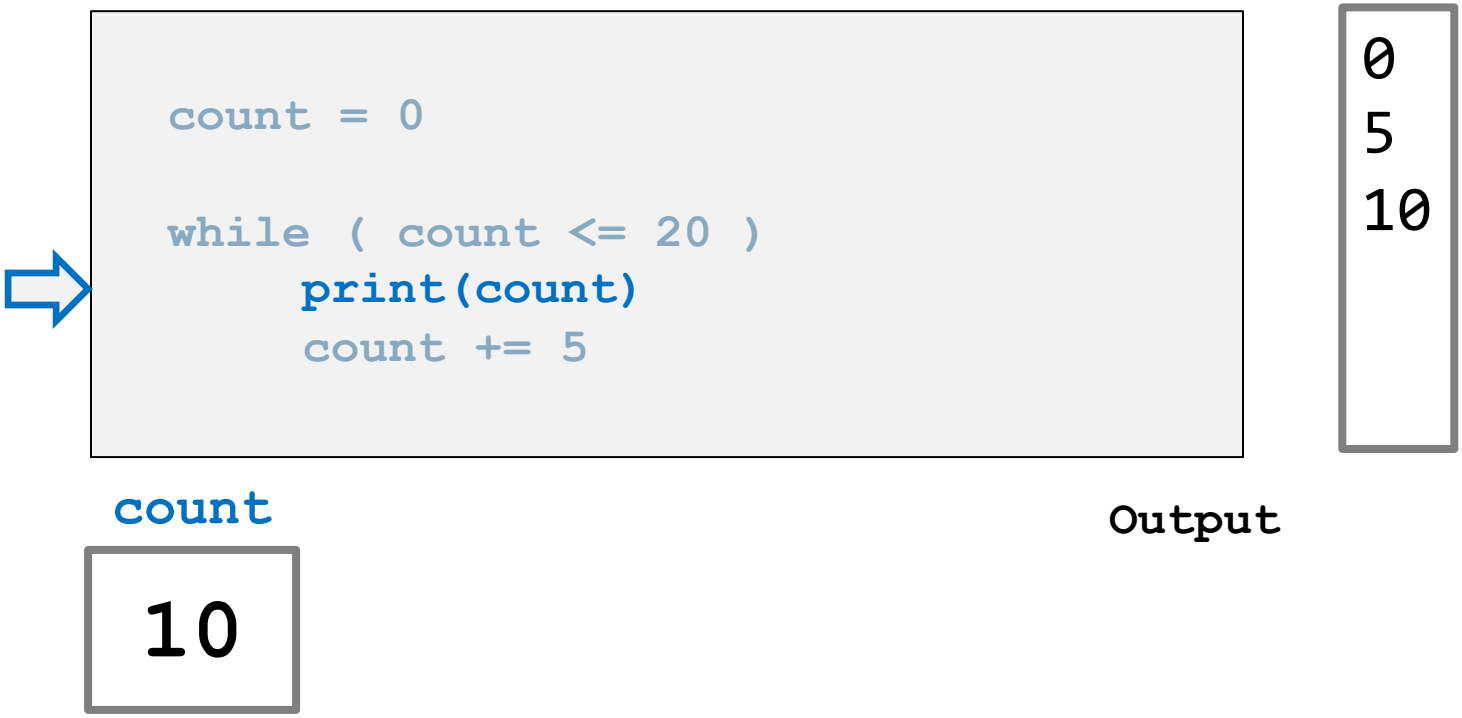
Execution of a **`while`** Loop

- **Control statement** checks if **`count`** is ***less or equal*** to 20 → **`true`** → executes the body of the loop.



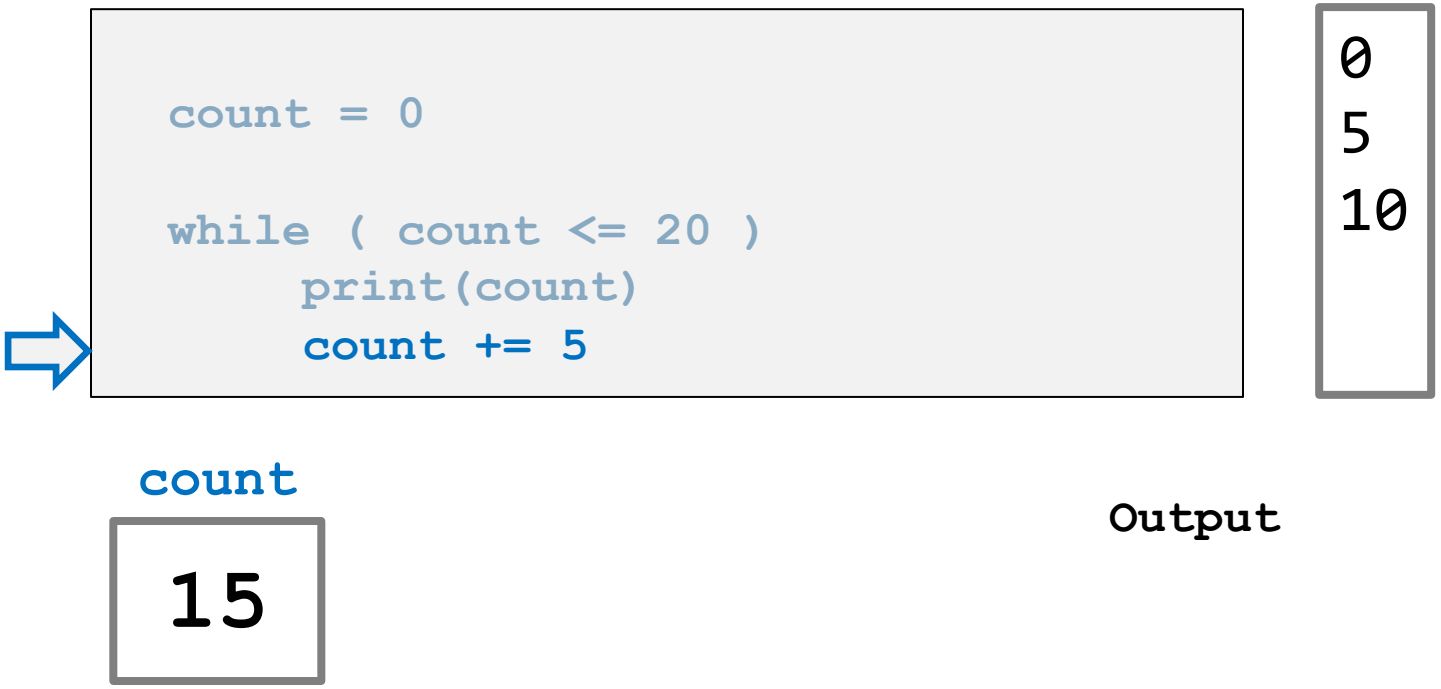
Execution of a **`while`** Loop

- First statement is executed and prints out 10.



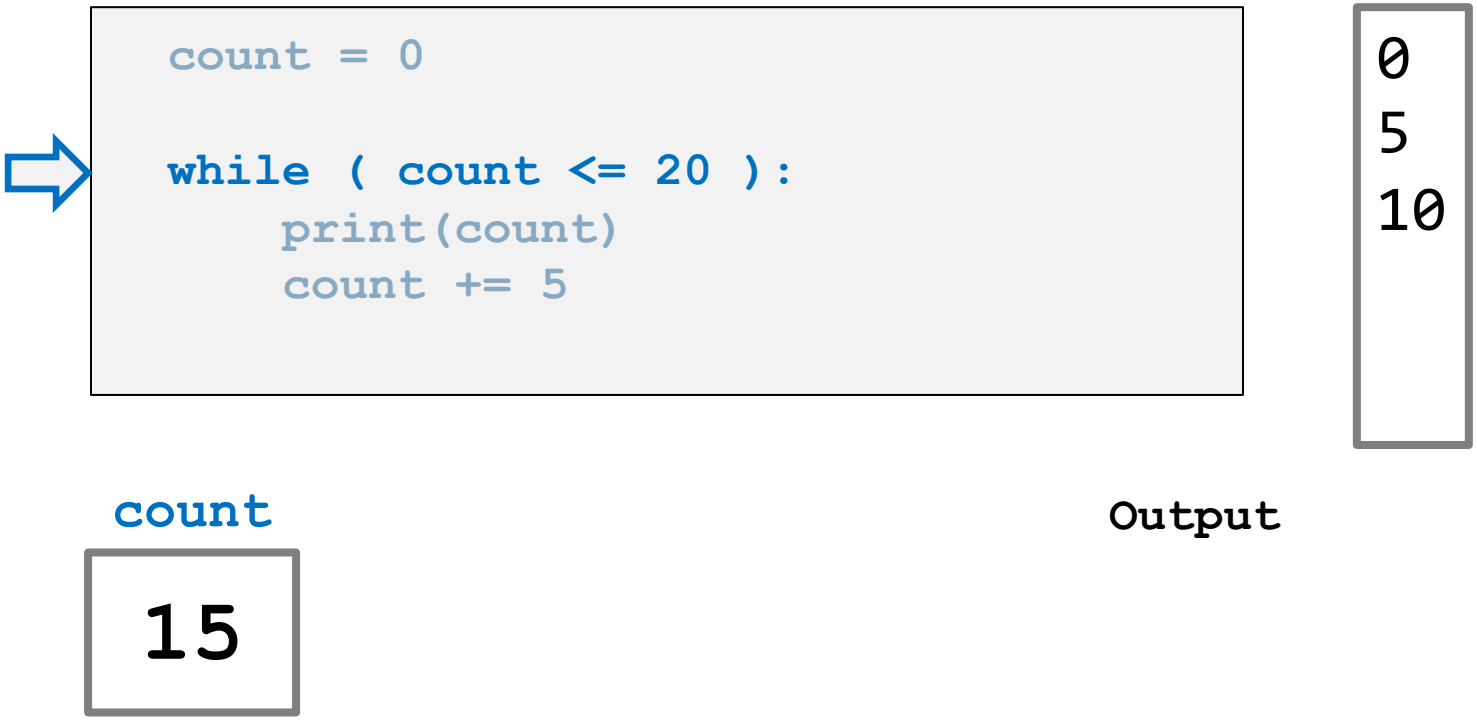
Execution of a **`while`** Loop

- Second statement is executed and increases the value of the variable **`count`** by 5.



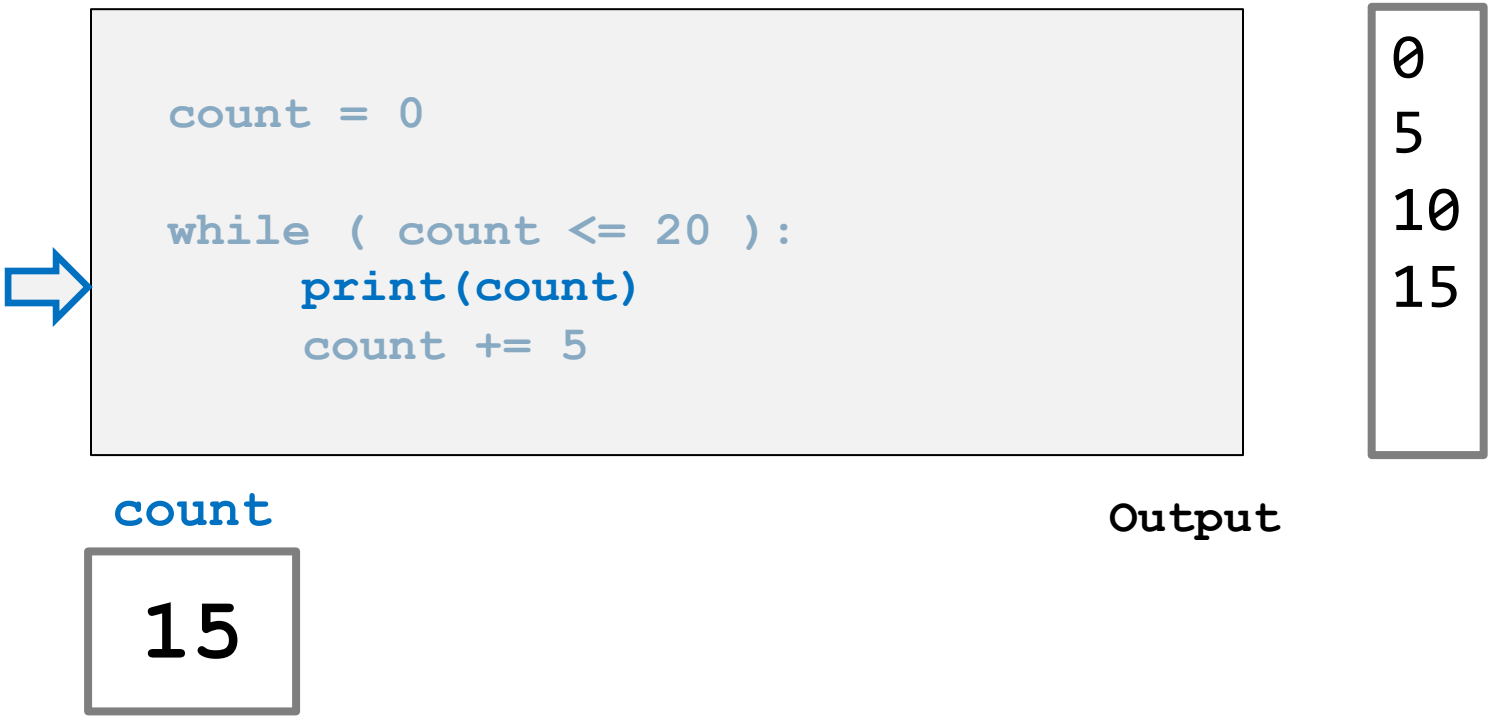
Execution of a **`while`** Loop

- **Control statement** checks if **`count`** is ***less or equal*** to 20 → **`true`** → executes the body of the loop.



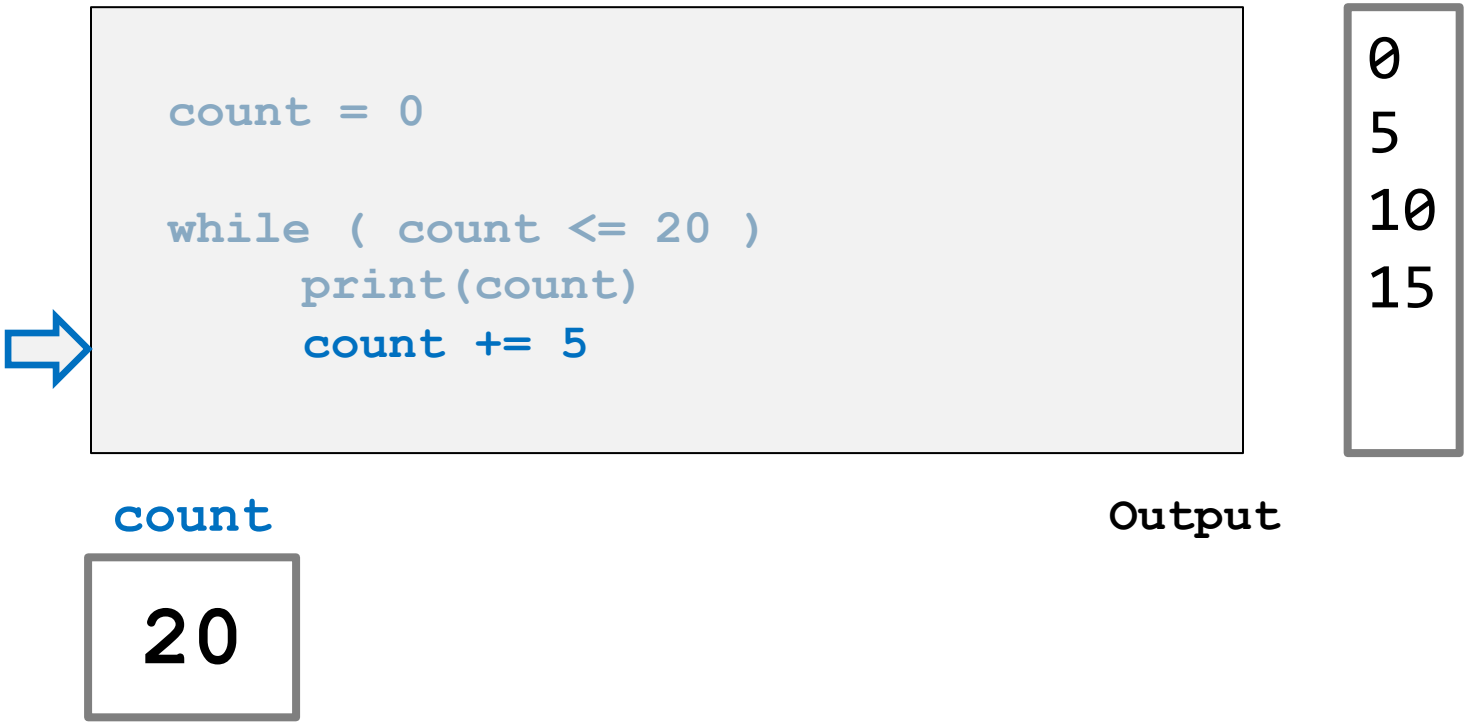
Execution of a **`while`** Loop

- First statement is executed and prints out 15.



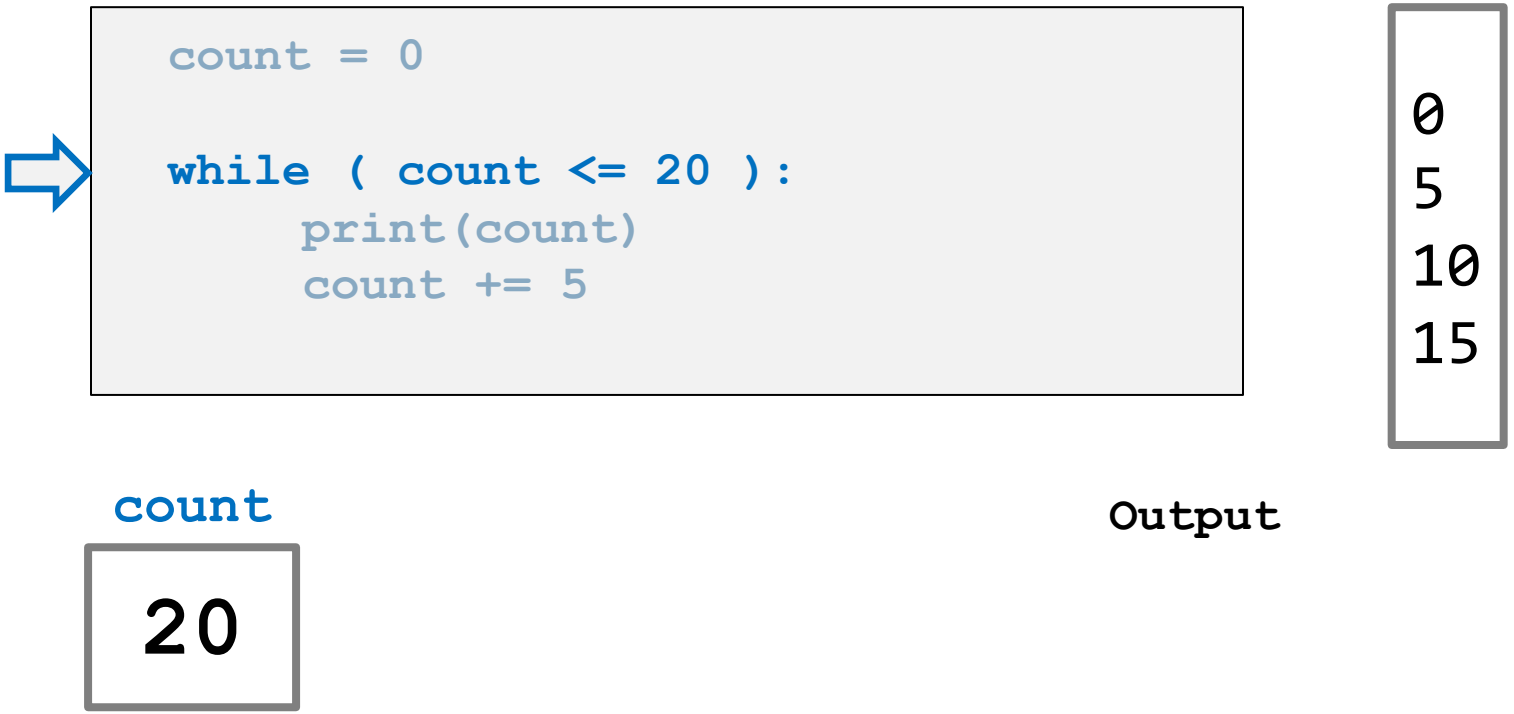
Execution of a **`while`** Loop

- Second statement is executed and increases the value of the variable **`count`** by 5.



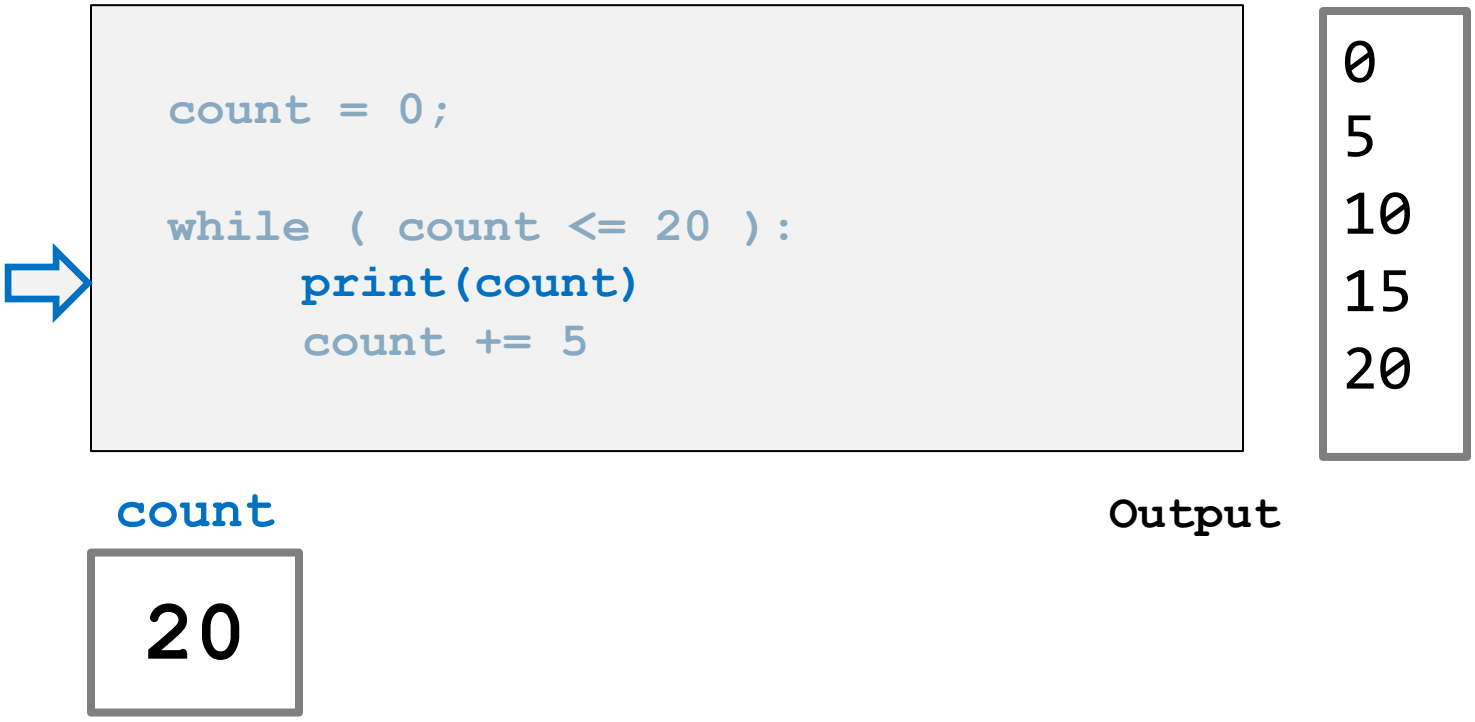
Execution of a **`while`** Loop

- **Control statement** checks if **`count`** is ***less or equal*** to 20 → **`true`** → executes the body of the loop.



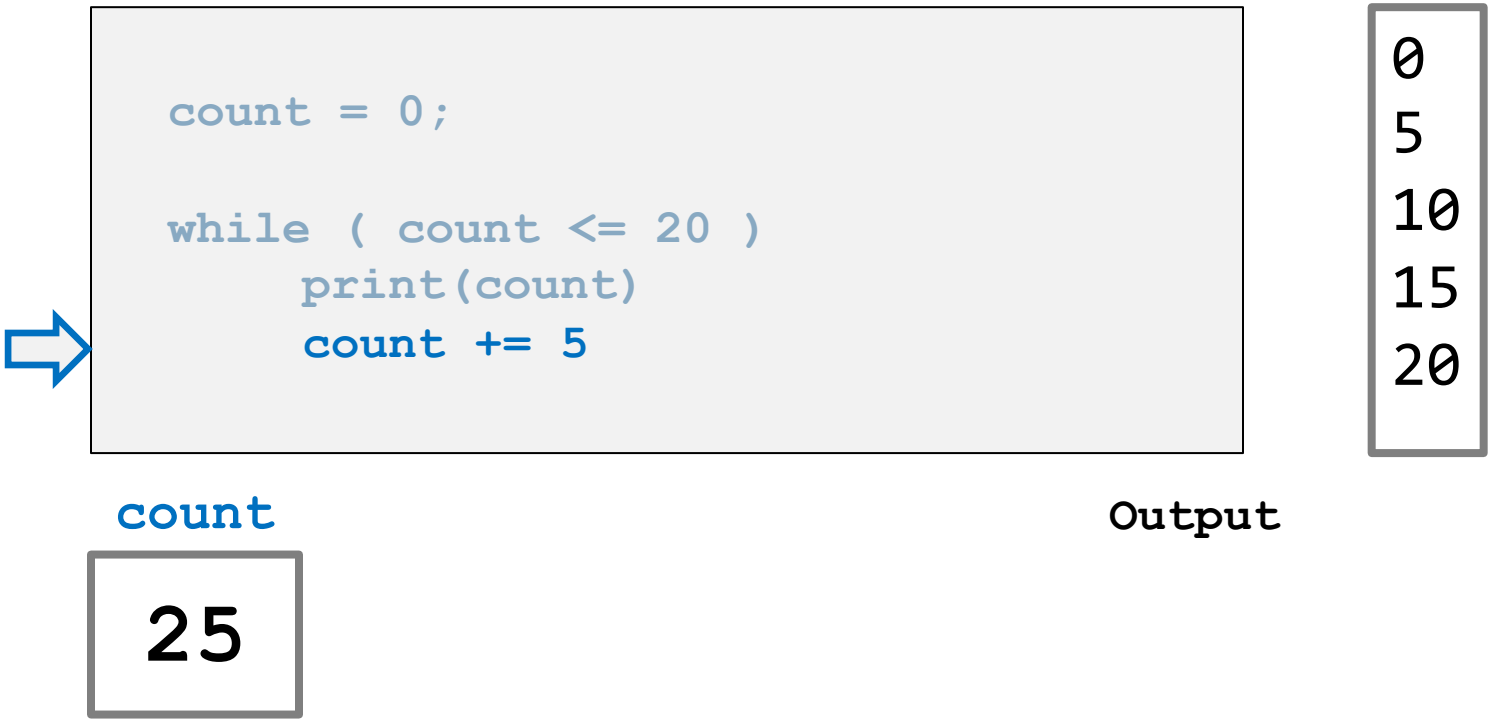
Execution of a **`while`** Loop

- First statement is executed and prints out 20.



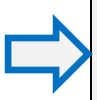
Execution of a **`while`** Loop

- Second statement is executed and increases the value of the variable **`count`** by 5.



Execution of a **`while`** Loop

- **Control statement** checks if **`count`** is ***less or equal*** to 20 → **`false`** → done with the loop.



```
count = 0

while ( count <= 20 ):
    print(count)
    count += 5
```

```
0
5
10
15
20
```

`count`

```
25
```

`Output`

Infinite Loops

- A **loop** that repeats without ever ending is called an ***infinite loop***.
  - What happens in our previous example if we do **not** update the variable `count`?

```
count = 0

while ( count <= 20 ):
    print(count)
    // count += 5
```

The variable `count` will always be 0 and will never reach 20.

Event-Controlled Loops

```
INITIAL_BALANCE = 50.00                  # Set the CONSTANTS
TARGET = 100.00
RATE = .5
balance = INITIAL_BALANCE                # Initialize the variable

while balance <= TARGET:                 # Check the loop variable
  year = year + 1
  balance = balance + balance * RATE # Update the loop variable
```

`while` Loop Examples

Loop	Output	Explanation

`i = 0`
`total = 0`
`while total < 10 :`
`  i = i + 1`
`  total = total + i`
`  print(i, total)`
	`1 1`
	`2 3`
	`3 6`
	`4 10`
		When `total` is 10, the loop condition is false, and the loop ends.

`i = 0`
`total = 0`
`while total < 10 :`
`  i = i + 1`
`  total = total - 1`
`  print(i, total)`
	`1 -1`
	`2 -3`
	`3 -6`
	`4 -10`
	`. . .`
		Because `total` never reaches 10, this is an "infinite loop" (see Common Error 4.2 on page 161).

`i = 0`
`total = 0`
`while total < 0 :`
`  i = i + 1`
`  total = total - i`
`  print(i, total)`
	(No output)	The statement `total < 0` is false when the condition is first checked, and the loop is never executed.

`while` Loop Examples (2)

Loop	Output	Explanation

`i = 0`
`total = 0`
`while total >= 10 :`
`  i = i + 1`
`  total = total + i`
`  print(i, total)`
	(No output)	The programmer probably thought, "Stop when the sum is at least 10." However, the loop condition controls when the loop is executed, not when it ends (see Common Error 4.2 on page 161).

`i = 0`
`total = 0`
`while total >= 0 :`
`  i = i + 1`
`  total = total + i`
`print(i, total)`
	(No output, program does not terminate)	Because `total` will always be greater than or equal to 0, the loop runs forever. It produces no output because the `print` function is outside the body of the loop, as indicated by the indentation.

Common Error: Incorrect Test Condition

- The loop body will only execute if the test condition is True.
- If bal is initialized as less than the TARGET and should grow until it reaches TARGET
  - Which version will execute the loop body?

```
while bal >= TARGET :
    year = year + 1
    interest = bal * RATE
    bal = bal + interest
```

```
while bal < TARGET :
    year = year + 1
    interest = bal * RATE
    bal = bal + interest
```

Common Error: Infinite Loops

- The loop body will execute until the test condition becomes False.
- What if you forget to update the test variable?
  - bal is the test variable (TARGET doesn't change)
  - You will loop forever! (or until you stop the program)

```
while bal < TARGET :
    year = year + 1
    interest = bal * RATE
```

Common Error: Off-by-One Errors

- A ‘counter’ variable is often used in the test condition
- Your counter can start at 0 or 1, but programmers often start a counter at 0
- If I want to paint all 5 fingers on one hand, when I am done?
  - If you start at 0, use “<“
  - 0, 1, 2, 3, 4

If you start at 1, use “<=“

1, 2, 3, 4, 5

```
finger = 0
FINGERS = 5
while finger < FINGERS :
    # paint finger
    finger = finger + 1
```

```
finger = 1
FINGERS = 5
while finger <= FINGERS :
    # paint finger
    finger = finger + 1
```

Summary of the **while** Loop

- `while` loops are very common
- Initialize variables before you test
  - The condition is tested BEFORE the loop body
    - This is called pre-test
    - The condition often uses a counter variable
  - Something inside the loop should change one of the variables used in the test
- Watch out for infinite loops!

```
num = 0;
while (num < 10):
    print(n)
    n += 1
```

How many times does this loop repeat?

A. 0
B. 9
C. 10
D. 11
E. infinite loop

```
num = 0
while (num > 10):
    print(num)
    n += 1
```

How many times does this loop repeat?

A. 0
B. 9
C. 10
D. 11
E. infinite loop

```
num = 1
while (num < 10):
    num = 2 * num
print(num)
```

What is the output?

A. 1 2 4 6 8
B. 2 4 6 8 10
C. 16
D. 2 4 8
E. None of these

Sentinel Values

Processing Sentinel Values

- Sentinel values are often used:
  - When you don't know how many items are in a list, use a 'special' character or value to signal the "last" item
  - For numeric input of positive numbers, it is common to use the value -1

A sentinel value denotes the end of a data set, but it is not part of the data.

```
salary = 0.0
while salary >= 0 :
   salary = float(input())
      if salary >= 0.0 :
         total = total + salary
         count = count + 1
```

Example – try it ? Think nested!

- Sample screen output:

```
Enter all the scores to be averaged.
Enter a negative number after you are done.
90
70
80
70
70
80
90
90
80
-1

The average is 80.0
```

Averaging a Set of Values – Ch 04. LAB 1

- Declare and initialize a ‘grades_total’ variable to 0
- Declare and initialize a ‘count’ variable to 0
- Declare and initialize a ‘grade’ variable to 0
- Prompt user with instructions
- Loop until sentinel value is entered
  - Save entered value to input variable (‘grade’)
  - If grade is not -1 or less (sentinel value)
    - Add grade variable to grades_ total variable
    - Add 1 to count variable
- Make sure you have at least one entry before you divide!
  - Divide total by count and output.
  - Done!

sentinel_example.py (1)

```
5   # Initialize variables to maintain the running total and count.
6   total = 0.0
7   count = 0
8
9   # Initialize salary to any non-sentinel value.
10  salary = 0.0

13  while salary >= 0.0 :

14      salary = float(input("Enter a salary or -1 to finish: "))
15      if salary >= 0.0 :

16          total = total + salary
17          count = count + 1
```

Outside the while loop: declare and initialize variables to use

Since `salary` is initialized to 0, the while loop statements will execute at least once

Input new `salary` and compare to sentinel

Update running `total` and `count` (to calculate the average later)

sentinel_example.py (2)

```
19  # Compute and print the average salary.
20  if count > 0 :
21      average = total / count
22      print("Average salary is", average)

23  else :
24      print("No data was entered.")
```

Prevent divide by 0

Calculate and output the average salary using the `total` and `count` variables

Program Run

```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

Priming Read

- Some programmers don't like the "trick" of initializing the input variable with a value other than a sentinel.

```
# Set salary to a value to ensure that the loop
# executes at least once.
salary = 0.0
while salary >= 0 :
```

- An alternative is to change the variable with a read before the loop.

```
salary = float(input("Enter a salary or -1 to finish: "))
while salary >= 0 :
```

Modification Read

- The input operation at the bottom of the loop is used to obtain the next input.

```
# Priming read
salary = float(input("Enter a salary or -1 to finish: "))
while salary >= 0.0 :
    total = total + salary
    count = count + 1
    # Modification read
    salary = float(input("Enter a salary or -1 to finish:"))
```

Boolean Variables and Sentinels

- A boolean variable can be used to control a loop
  - Sometimes called a ‘flag’ variable

```
done = False
while not done :
    value = float(input("Enter a salary or -1 to finish: "))
    if value < 0.0:
        done = True
    else :
        # Process value
```

Initialize done so that the loop will execute

Set done ‘flag’ to True if sentinel value is found

Common Loop Algorithms

Common Loop Algorithms

1. Sum and Average Value
2. Counting Matches
3. Prompting until a Match Is Found
4. Maximum and Minimum
5. Comparing Adjacent Values

Sum Example

- Sum of Values
  - Initialize total to 0
  - Use while loop with sentinel

```
total = 0.0
user_value = input("Enter value: ")
while user_value != "" :
    value = float(user_value)
    total = total + value
    user_value = input("Enter value: ")
```

Counting Matches (e.g., Negative Numbers)

- Counting Matches
  - Initialize negatives to 0
  - Use a while loop
  - Add to negatives per match

```
negatives = 0
user_value = input("Enter value: ")
while inputStr != "" :
    value = int(user_value)
    if value < 0 :
        negatives = negatives + 1
    user_value = input("Enter value:")

print("There were", negatives,
"negative values.")
```

Prompt Until a Match is Found

- Initialize boolean flag to False
- Test sentinel in while loop
  - Get input, and compare to range
    - If input is in range, change flag to True
    - Loop will stop executing

```
valid = False
while not valid :
   value = int(input("Please enter a positive value < 100: "))
   if value > 0 and value < 100 :
      valid = True
   else :
      print("Invalid input.")
```

This is an excellent way to validate use provided inputs

Maximum

- Get first input value
  - By definition, this is the largest that you have seen so far
- Loop while you have a valid number (non-sentinel)
  - Get another input value
  - Compare new input to largest (or smallest)
  - Update largest if necessary

```
largest = int(input("Enter a value: "))
next = input("Enter a value: ")
while next != "" :
    value = int(next)
    if value > largest :
        largest = value
    next = input("Enter a value: ")
```

Minimum

- Get first input value
  - This is the smallest that you have seen so far!
- Loop while you have a valid number (non-sentinel)
  - Get another input value
  - Compare new input to largest (or smallest)
  - Update smallest if necessary

```
smallest = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "“ :
   value = int(inputStr)
   if value < smallest :
      smallest = value
   inputStr = input("Enter a value: ")
```

Comparing Adjacent Values

- Get first input value
- Use `while` to determine if there are more to check
  - Copy input to previous variable
  - Get next value into input variable
  - Compare input to previous, and output if same

```
value = int(input("Enter a value: "))
next = input("Enter a value: ")
while next != "" :
   previous = value
   value = int(next)
   if value == previous :
      print("Duplicate input")
   next = input("Enter a value: ")
```

The **for** Loop

The `for` Loop

- Uses of a **`for`** loop:
  - The **`for`** loop can be used to iterate over the contents of any **container**.
  - A **container** is is an object (Like a **string**) that contains or stores a collection of elements
  - A **string** is a container that stores the collection of characters in the string

while compared to **for**

- Note an important difference between the while loop and the for loop.
- In the while loop, the *index variable* i is assigned 0, 1, and so on.
- In the for loop, the *element variable* is assigned stateName[0], stateName[1], and so on.

while version

```
stateName = "Virginia"
i = 0
while i < len(stateName) :
   letter = stateName[i]
   print(letter)
   i += 1
```

for version

```
stateName = "Virginia"
for letter in stateName :
   print(letter)
```

while compared to **for**

- Uses of a for loop:
  - A for loop can also be used as a count-controlled loop that iterates over a range of integer values.

`while` version

```
i = 1
while i < 10 :
    print(i)
    i += 1
```

```
i = 0
while i < 10 :
    print(i)
    i += 1
```

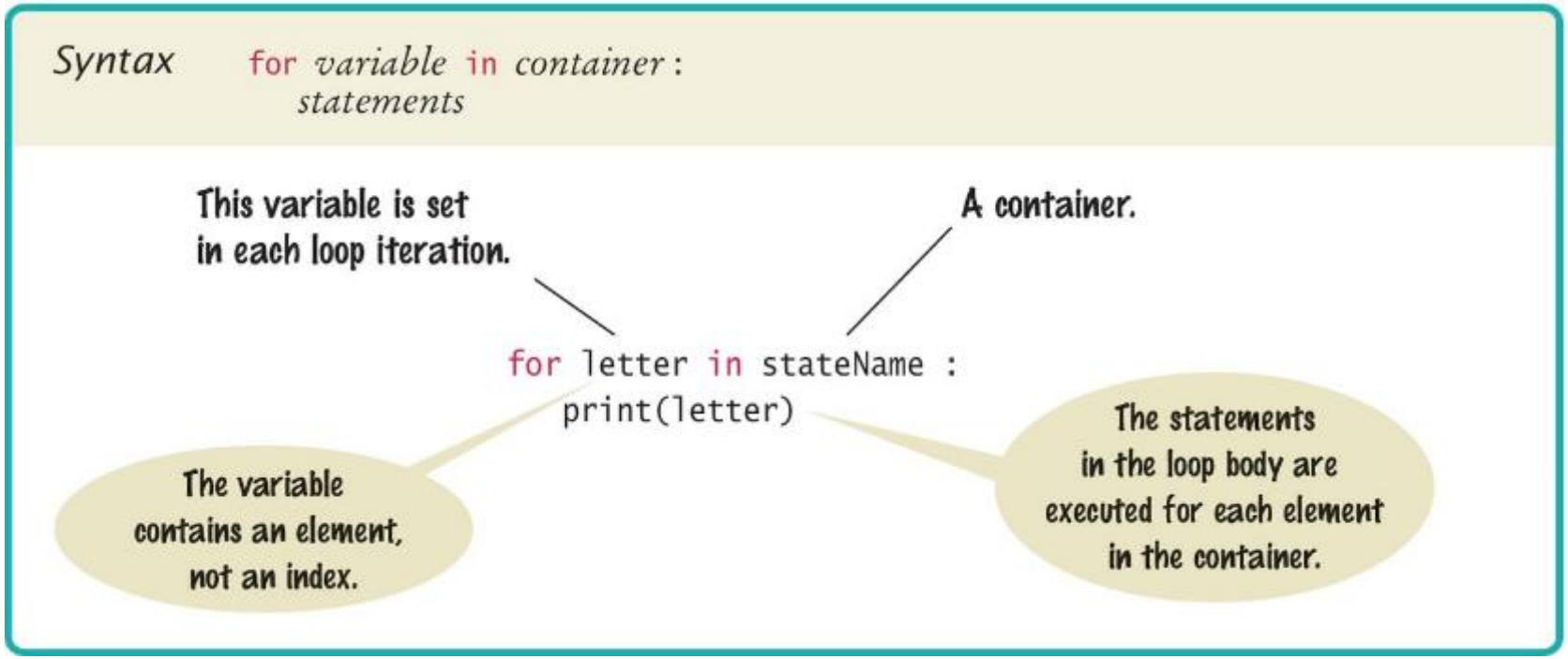
`for` version

```
for i in range(1, 10) :
    print(i)
```

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

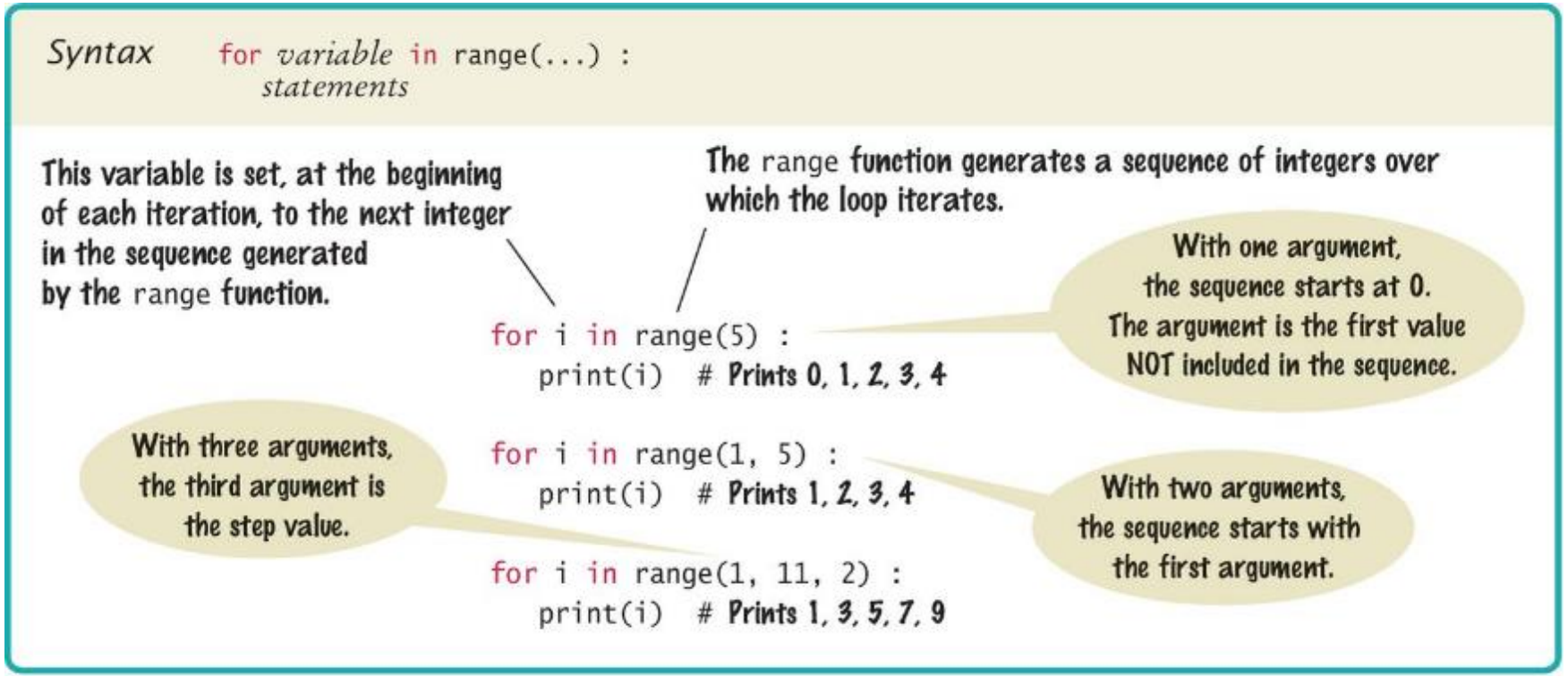
Syntax of a **for** Statement (Container)

- Using a **for** loop to iterate over the contents of a container, an element at a time.



Syntax of a **for** Statement: range()

- for loop as a count-controlled loop to iterate over a range of integer values
- the range function is an ***overloaded*** function: range(int), range(int, int), range(int, int, int),



for loops in range()

Table 2 for Loop Examples

Loop	Values of i	Comment
`for i in range(6) :`	0, 1, 2, 3, 4, 5	Note that the loop executes 6 times.
`for i in range(10, 16) :`	10, 11, 12, 13, 14 15	The ending value is never included in the sequence.
`for i in range(0, 9, 2) :`	0, 2, 4, 6, 8	The third argument is the step value.
`for i in range(5, 0, -1) :`	5, 4, 3, 2, 1	Use a negative step value to count down.

Investment Example

```
1  ##
2  #  This program prints a table showing the growth of an investment.
3  #
4
5  # Define constant variables.
6  RATE = 5.0
7  INITIAL_BALANCE = 10000.0
8
9  # Obtain the number of years for the computation.
10 numYears = int(input("Enter number of years: "))
11
12 # Print the table of balances for each year.
13 balance = INITIAL_BALANCE
14 for year in range(1, numYears + 1) :
15    interest = balance * RATE / 100
16    balance = balance + interest
17    print("%4d %10.2f" % (year, balance))
```

Steps to Writing a Loop

- Planning:
  - Decide what work to do inside the loop
  - Specify the loop condition
  - Determine loop type
  - Setup variables before the first loop
  - Process results when the loop is finished
  - Trace the loop with typical examples
- Coding:
  - Implement the loop in Python

A Special Form of the **`print`** Function

- Python provides a special form of the print function that does not start a new line after the arguments are displayed
- This is used when we want to print items on the same line using multiple print statements
- For example the two statements:

```
print(“00”, end=””)
print(3 + 4)
```

- Produce the output:

```
007
```

- Including **end=“”** as the last argument to the print function prints an empty string after the arguments, instead on a new line
- The output of the next **print** function starts on the same line

Nested Loops

Loops Inside of Loops

- In Chapter Three we learned how to nest **if** statements to allow us to make complex decisions
  - Remember that to nest the **if** statements we need to indent the code block
- Complex problems sometimes require a nested loop, one loop nested inside another loop
  - The nested loop will be indented inside the code block of the first loop
- A good example of using nested loops is when you are processing cells in a table
  - The outer loop iterates over all of the rows in the table
  - The inner loop processes the columns in the current row

Nested Loop Examples

Table 3 Nested Loop Examples

Nested Loops	Output	Explanation

`for i in range(3) :`
`  for j in range(4) :`
`  print("*", end="")`
`  print()`
	****
	****
	****
		Prints 3 rows of 4 asterisks each.

`for i in range(4) :`
`  for j in range(3) :`
`  print("*", end="")`
`  print()`
	***
	***
	***
	***
		Prints 4 rows of 3 asterisks each.

`for i in range(4) :`
`  for j in range(i + 1) :`
`  print("*", end="")`
`  print()`
	*
	**
	***
	****
		Prints 4 rows of lengths 1, 2, 3, and 4.

Nested Loop Examples (2)

Table 3 Nested Loop Examples

`for i in range(3) :`
`  for j in range(5) :`
`  if j % 2 == 1 :`
`  print("*", end="")`
`  else :`
`  print("-", end="")`
`  print()`
	-*-*-
	-*-*-
	-*-*-
		Prints alternating dashes and asterisks.

`for i in range(3) :`
`  for j in range(5) :`
`  if i % 2 == j % 2 :`
`  print("*", end="")`
`  else :`
`  print(" ", end="")`
`  print()`
	* * *
	 * *
	* * *
		Prints a checkerboard pattern.

Processing Strings

Processing Strings

- A common use of loops is to process or evaluate strings.
- For example, you may need to count the number of occurrences of one or more characters in a string or verify that the contents of a string meet certain criteria.

Counting Matches

- Suppose you need to count the number of uppercase letters contained in a string.
- We can use a for loop to check each character in the string to see if it is upper case
- The loop below sets the variable **char** equal to each successive character in the string
- Each pass through the loop tests the next character in the string to see if it is uppercase

```
uppercase = 0
for char in string :
   if char.isupper() :
      uppercase += 1
```

Counting Vowels

- Suppose you need to count the vowels within a string
- We can use a for loop to check each character in the string to see if it is in the string of vowels “aeiuo”
- The loop below sets the variable **char** equal to each successive character in the string
- Each pass through the loop tests the lower case of the next character in the string to see if it is in the string “aeiou”

```
vowels = 0
for char in word :
   if char.lower() in "aeiou" :
      vowels += 1
```

Finding All Matches Example

- When you need to examine every character in a string, independent of its position we can use a for statement to examine each character
- If we need to print the position of each uppercase letter in a sentence we can test each character in the string and print the position of all uppercase characters
- We set the range to be the length of the string
  - We test each character
  - If it is uppercase we print its position in the string

```
sentence = input("Enter a sentence: ")
for i in range(len(sentence)) :
   if sentence[i].isupper() :
      print(i)
```

Finding the First Match

- This example finds the position of the first digit in a string.

```
found = False
position = 0
while not found and position < len(string):
   if string[position].isdigit():
      found = True
   else:
      position = position + 1

if found:
   print("First digit occurs at position", position)
else:
   print("The string does not contain a digit.")
```

Finding the Last Match

- Here is a loop that finds the position of the last digit in the string.
- This approach uses a while loop to start at the last character in a string and test each value moving from the end of the string to the start of the string
  - Position is set to the length of the string - 1
  - If the character is not a digit, we decrease position by 1
  - Until we find a digit, or process all the characters

```
found = False
position = len(string) - 1
while not found and position >= 0:
   if string[position].isdigit():
      found = True
   else:
      position = position - 1
```

Building a New String

- One of the minor annoyances of online shopping is that many web sites require you to enter a credit card without spaces or dashes, which makes double-checking the number rather tedious.
- How hard can it be to remove dashes or spaces from a string?

Practice: Ch. 04 LAB 3

- Write a program that takes in a “credit card number” from a user
- Validate the entry:
  - Must have 16 digits
  - Allow for ”-” being included or not
- Give error message in the following cases and prompt the user to re-enter:
  - Non digit characters are included
  - Too many or too few digits are given

Application: Random Numbers and Simulations

Random Numbers/Simulations

- Games often use random numbers to make things interesting
  - Rolling Dice
  - Spinning a wheel
  - Pick a card
- A simulation usually involves looping through a sequence of events
  - Days
  - Events

Generating Random Numbers

- The Python library has a *random number generator* that produces numbers that appear to be random
  - The numbers are not completely random. The numbers are drawn from a sequence of numbers that does not repeat for a long time
  - random() returns a number that is >= 0 and < 1

Simulating Die Tosses

- Goal:
  - To generate a random integer in a given range we use the randint() function
  - Randint has two parameters, the range (inclusive) of numbers generated

ch04/dice.py

```
1  ##
2  #  This program simulates tosses of a pair of dice.
3  #
4
5  from random import randint
6
7  for i in range(10) :
8     # Generate two random numbers between 1 and 6, inclusive.
9     d1 = randint(1, 6)
10    d2 = randint(1, 6)
11
12    # Print the two values.
13    print(d1, d2)
```

Program Run

```
1 5
6 4
1 1
4 5
6 4
3 2
4 2
3 5
5 2
4 5
```

Problem Solving: Solve a Simpler Problem First

Simplify a Complex Problem

- As we learn more about programming, the complexity of the tasks we are solving increases
- When we are faced with a complex task we should apply a critical skill:
  - Simplifying the problem and solving the simpler problem first
- Our simplification (AKA problem decomposition) skills improve with practice