PGR107
Python Programming

Lecture 4 – Loops

Kristiania University College

Chapter 4 - Loops

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 process strings
- To use a computer for simulations

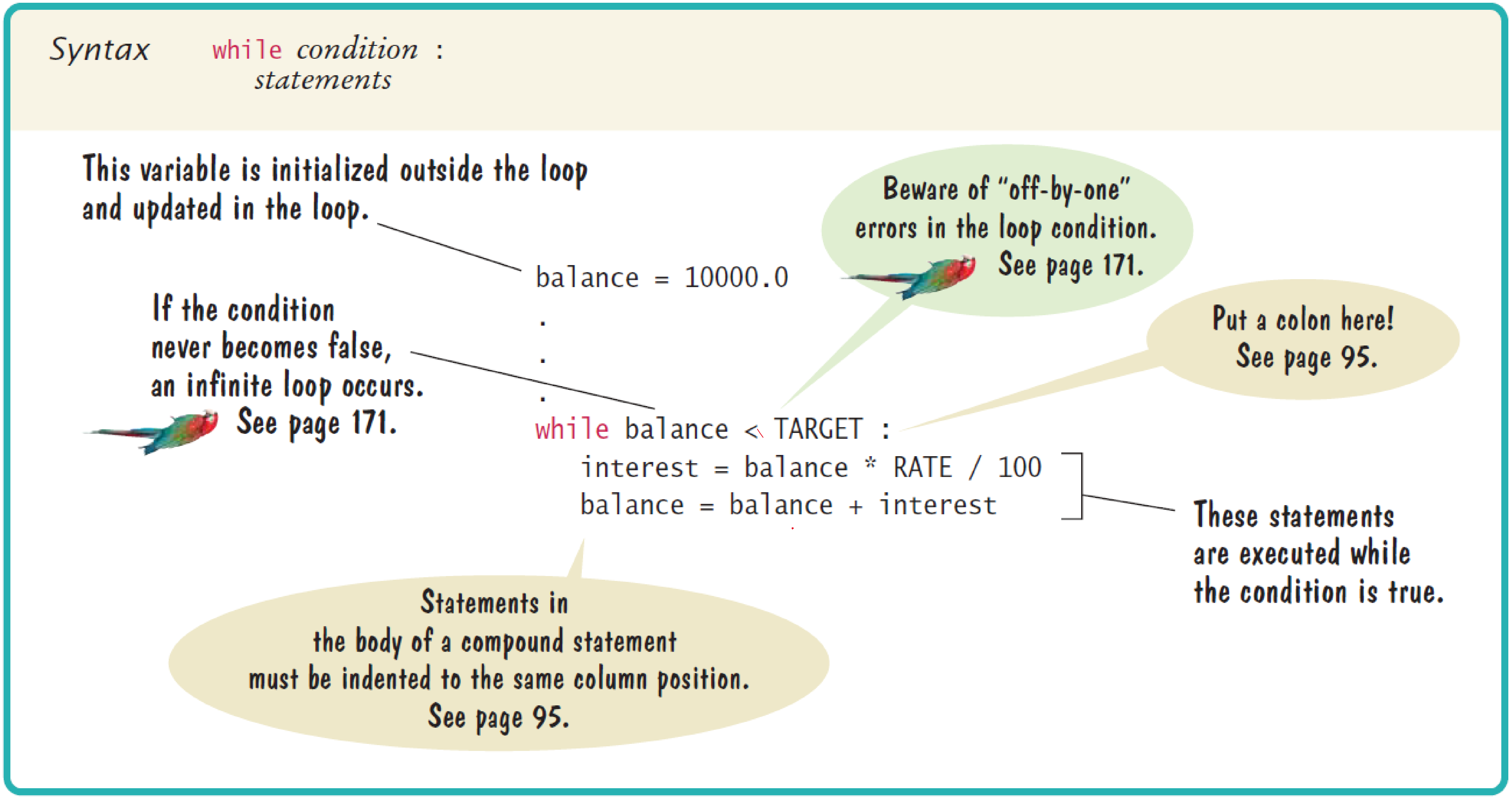
Loops

- In a loop, a part of a program is repeated over and over, until a specific goal is reached. Loops are important for calculations that require repeated steps and for processing input consisting of many data items.

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

- **Example:** Sum of integer numbers from 1 too 100

While Loop



Syntax `while` *condition* :
    statements

```
balance = 10000.0
.
.
.
while balance < TARGET :
    interest = balance * RATE / 100
    balance = balance + interest
```

This variable is initialized outside the loop and updated in the loop.

If the condition never becomes false, an infinite loop occurs. See page 171.

Beware of "off-by-one" errors in the loop condition. See page 171.

Put a colon here! See page 95.

These statements are executed while the condition is true.

Statements in the body of a compound statement must be indented to the same column position. See page 95.

While Loop Examples

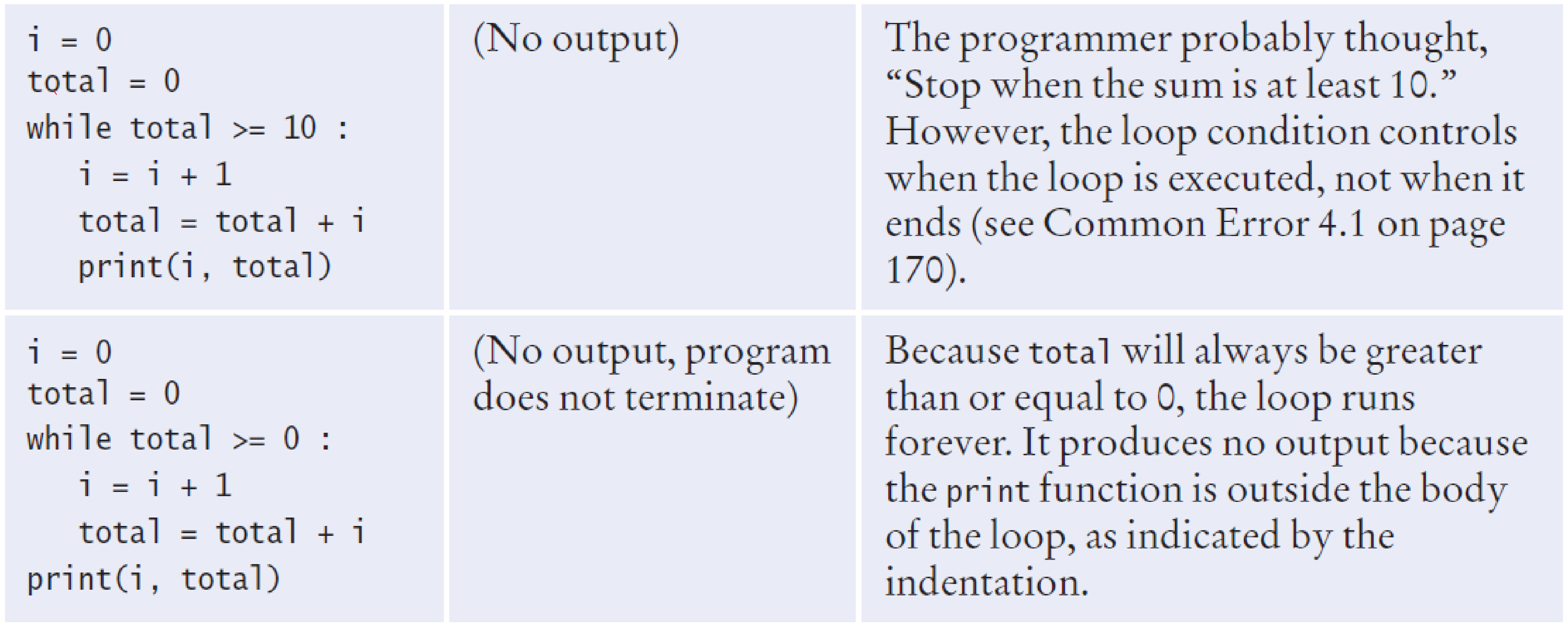
Loop	Output	Explanation

`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 171).

`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

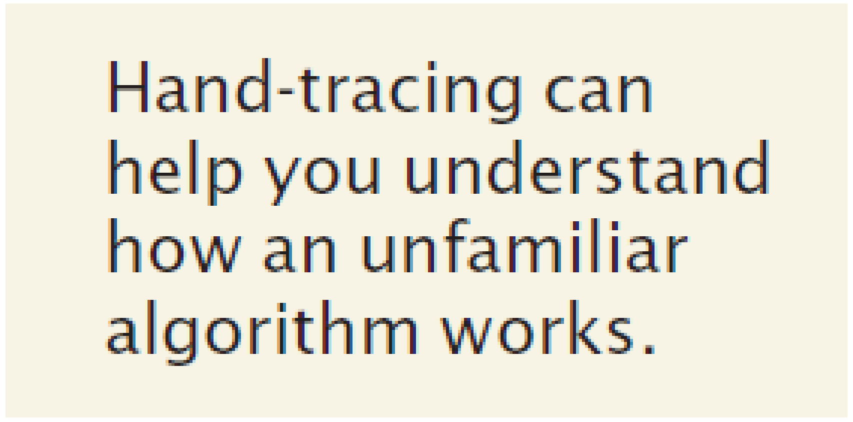


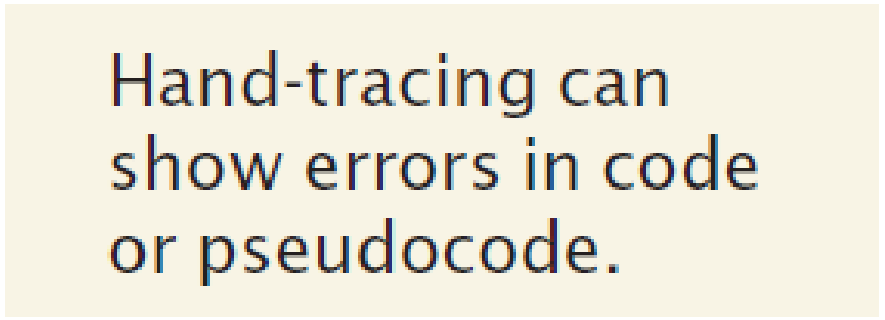
`i = 0`
`total = 0`
`while total >= 10 :`
`    i = i + 1`
`    total = total + i`
`    print(i, total)`
	(No output)	The programmer probably thought, "Stop when the sum is at least 10." However, the loop condition controls when the loop is executed, not when it ends (see Common Error 4.1 on page 170).

`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.

Hand-Tracing

- **Hand-tracing** is a simulation of code execution in which you step through instructions and track the values of the variables.


Hand-tracing can help you understand how an unfamiliar algorithm works.

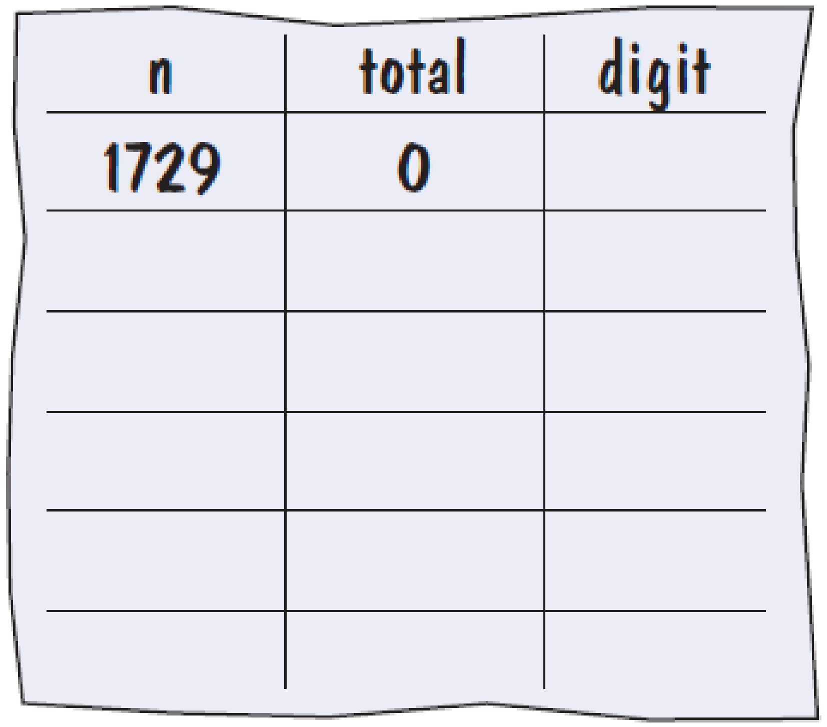

Hand-tracing can show errors in code or pseudocode.

Hand-Tracing

- Consider the following piece of code. What value is displayed?

```
n = 1729
total = 0
while n > 0 :
   digit = n % 10
   total = total + digit
   n = n // 10

print(total)
```



n	total	digit
1729	0	

Processing Sentinel Values

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

- Whenever you read a sequence of inputs, you need to have some method of indicating the end of the sequence. Sometimes you are lucky and no input value can be zero. Then you can prompt the user to keep entering numbers, or 0 to finish the sequence. If zero is allowed but negative numbers are not, you can use –1 to indicate termination.

- Such a value, which is not an actual input, but serves as a signal for termination, is called a **sentinel**.

Processing Sentinel Values

- Now consider the case in which any number (positive, negative, or zero) can be an acceptable input. In such a situation, you must use a sentinel that is not a number (such as the letter Q).

```
inputStr = input("Enter a value or Q to quit: ")
while inputStr != "Q" :
   value = float(inputStr)
   Process value.
   inputStr = input("Enter a value or Q to quit: ")
```

Processing Sentinel Values

- Sentinel values can also be processed using a **Boolean variable** for the loop termination:

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

- As an alternative, you can use the **break** statement:

```
while True :
   value = float(input("Enter a salary or -1 to finish: "))
   if value < 0.0 :
      break
   Process value.
```

Common Loop Algorithms

- **a) Sum and Average Value**

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

```
total = 0.0
count = 0
inputStr = input("Enter value: ")
while inputStr != "" :
   value = float(inputStr)
   total = total + value
   count = count + 1
   inputStr = input("Enter value: ")

if count > 0 :
   average = total / count
else :
   average = 0.0
```

Common Loop Algorithms

- **b) Counting Matches:** To count values that fulfill a condition, check all values and increment a counter for each match. For example, you may want to count how many negative values are included in a sequence of integers.

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

In a loop that counts matches, a counter is incremented whenever a match is found.

Common Loop Algorithms

- **c) Prompting Until a Match is Found:** Suppose you are asking the user to enter a positive value < 100:

```
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.")
```

Common Loop Algorithms

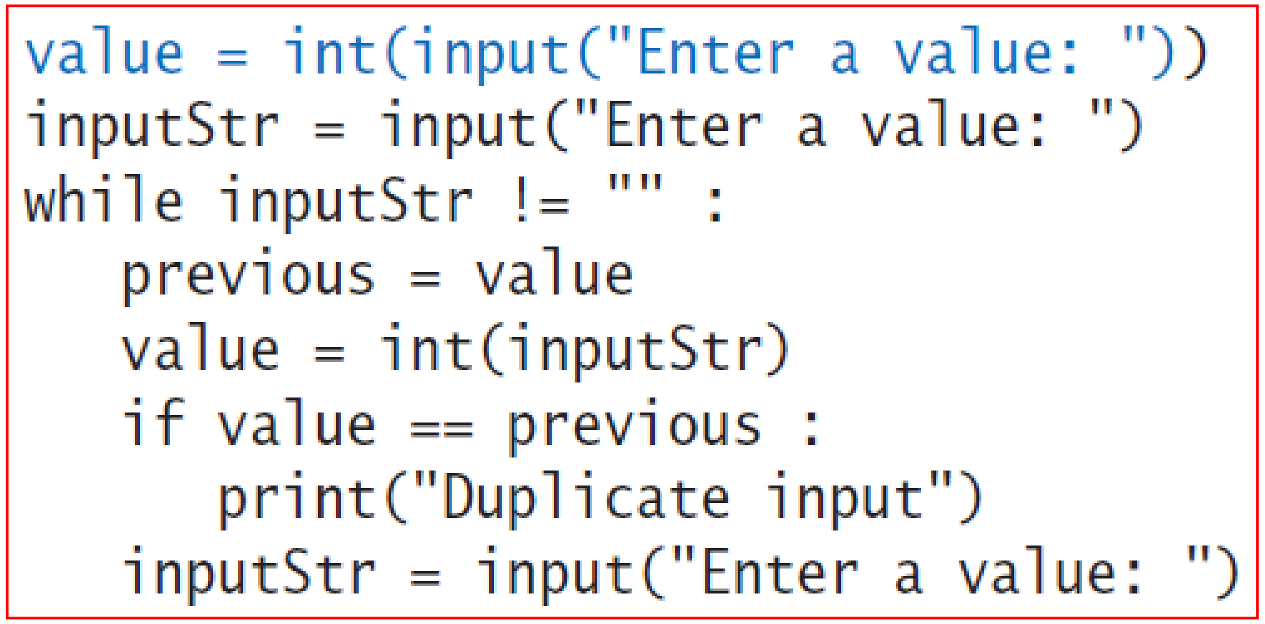
- **d) Maximum and Minimum:** To compute the largest/smallest value in a sequence, keep a variable that stores the largest/smallest element that you have encountered and update the variable when you find a larger/smaller one:

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

```
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: ")
```

Common Loop Algorithms

- **e) Comparing Adjacent Values:** When processing a sequence of values in a loop, you sometimes need to compare a value with the value that just preceded it. For example, suppose you want to check whether a sequence of inputs such as 1 7 2 9 9 4 9 contains adjacent duplicates.



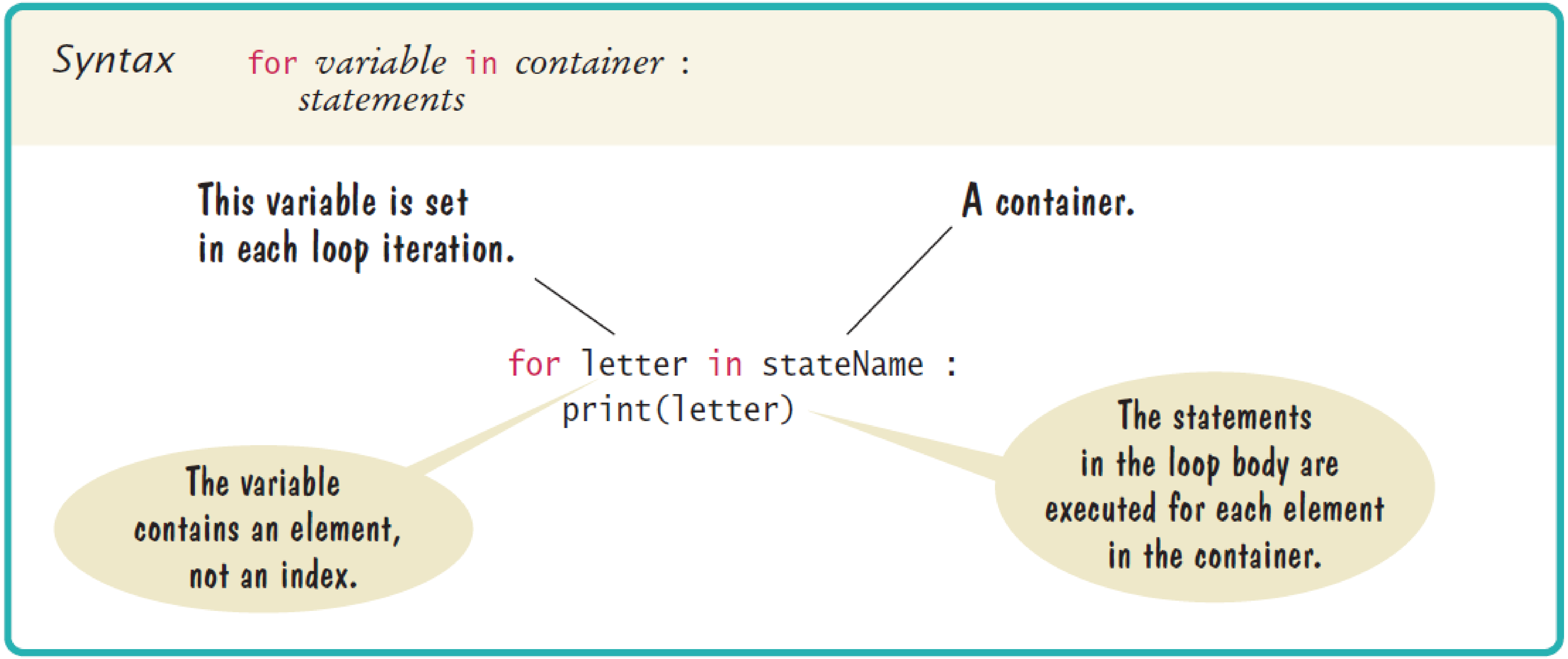
```
value = int(input("Enter a value: "))
inputStr = input("Enter a value: ")
while inputStr != "" :
   previous = value
   value = int(inputStr)
   if value == previous :
      print("Duplicate input")
   inputStr = input("Enter a value: ")
```

break Statement

- Immediately exits whatever loop it is in
- Skips remaining expressions in code block
- Exits only innermost loop

```
while <condition 1>:
      while <condition 2>:
            <expression 1>
            break
            <expression 2>
      <expression 3>
```

For Loop



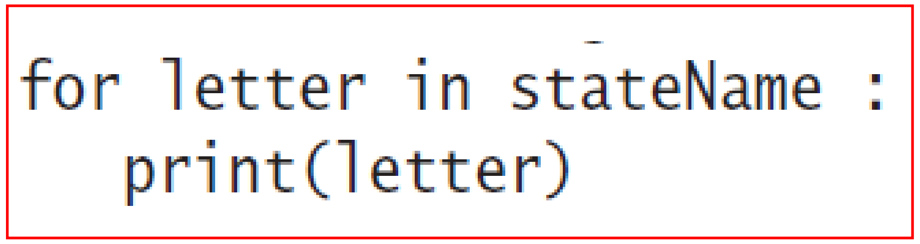
```
Syntax    for variable in container :
              statements
```

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

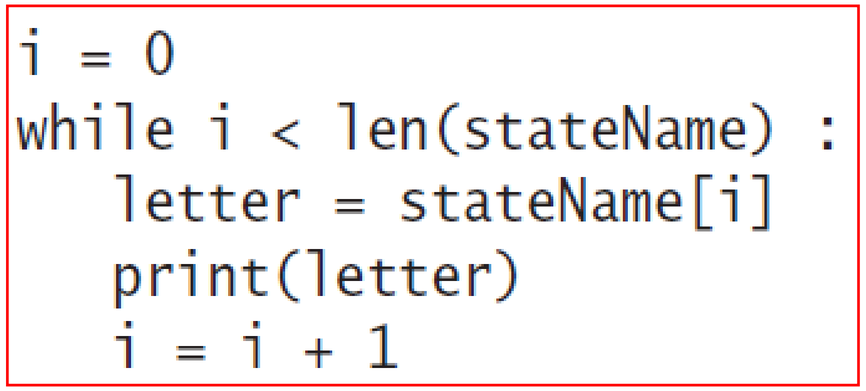
This variable is set in each loop iteration.
A container.
The variable contains an element, not an index.
The statements in the loop body are executed for each element in the container.

For Loop

- Suppose we want to print a string, with one character per line:
- **stateName = “Virginia”**



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



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

For Loop

- The for loop can be used with the range function to iterate over a range of integer values.

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

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

- The first argument of the range function is the first value in the sequence. Values are included in the sequence while the are less than the second argument.

For Loop

- **The *range* Function**: By default, the range function creates the sequence in steps of 1. This can be changed by including a step value as the third argument to the function:

```
for i in range(1, 10, 2) :     # i = 1, 3, 5, ..., 9
   print(i)
```

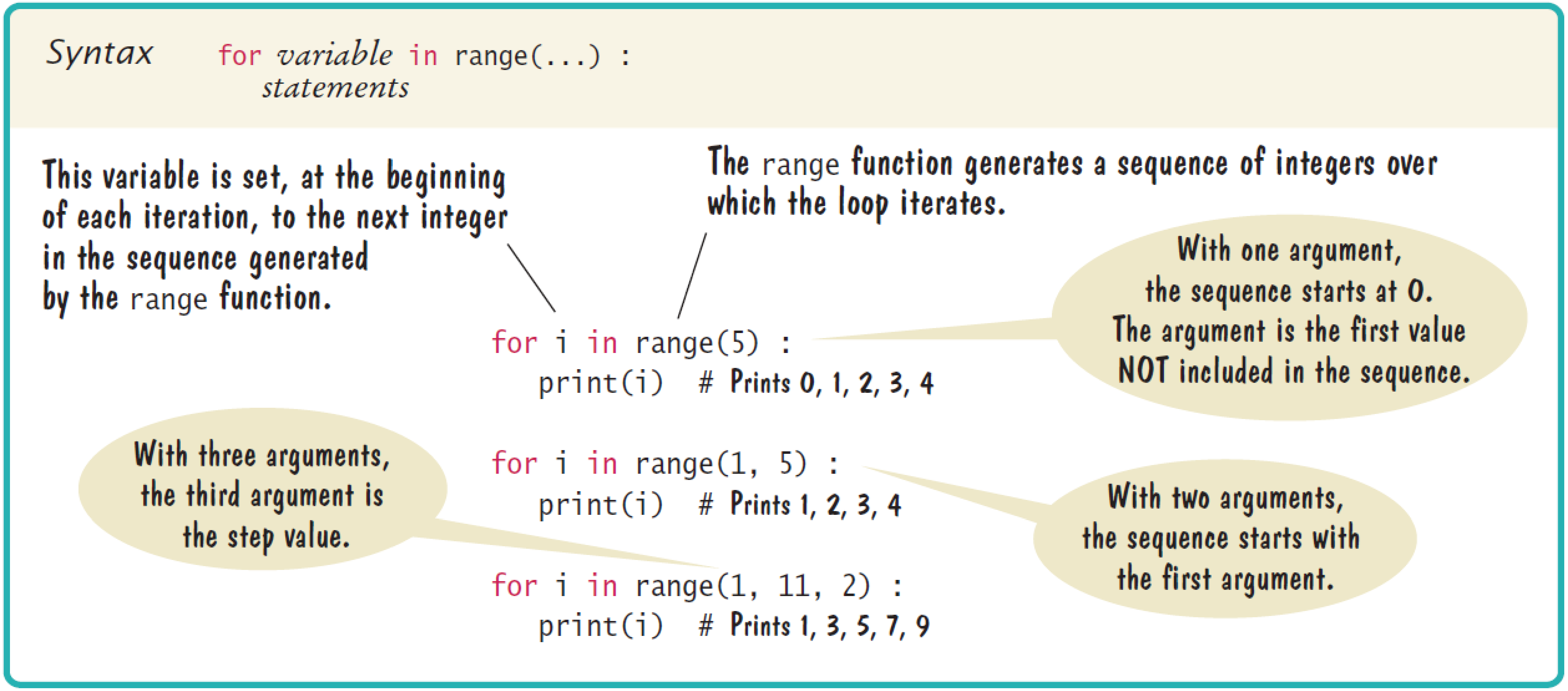
- We can also have the **for loop** count **down** instead of up:

```
for i in range(10, 0, -1) :    # i = 10, 9, 8, ..., 1
   print(i)
```

- We can use the **range function** with a **single argument**. When you do, the range of values starts at **zero**.

```
for i in range(10) :   # i = 0, 1, 2, ..., 9
   print("Hello")     # Prints Hello ten times
```

For Loop



Syntax

```
for variable in range(...) :
    statements
```

This variable is set, at the beginning of each iteration, to the next integer in the sequence generated by the range function.

The range function generates a sequence of integers over which the loop iterates.

```
for i in range(5) :
    print(i)   # Prints 0, 1, 2, 3, 4
```

With one argument, the sequence starts at 0. The argument is the first value NOT included in the sequence.

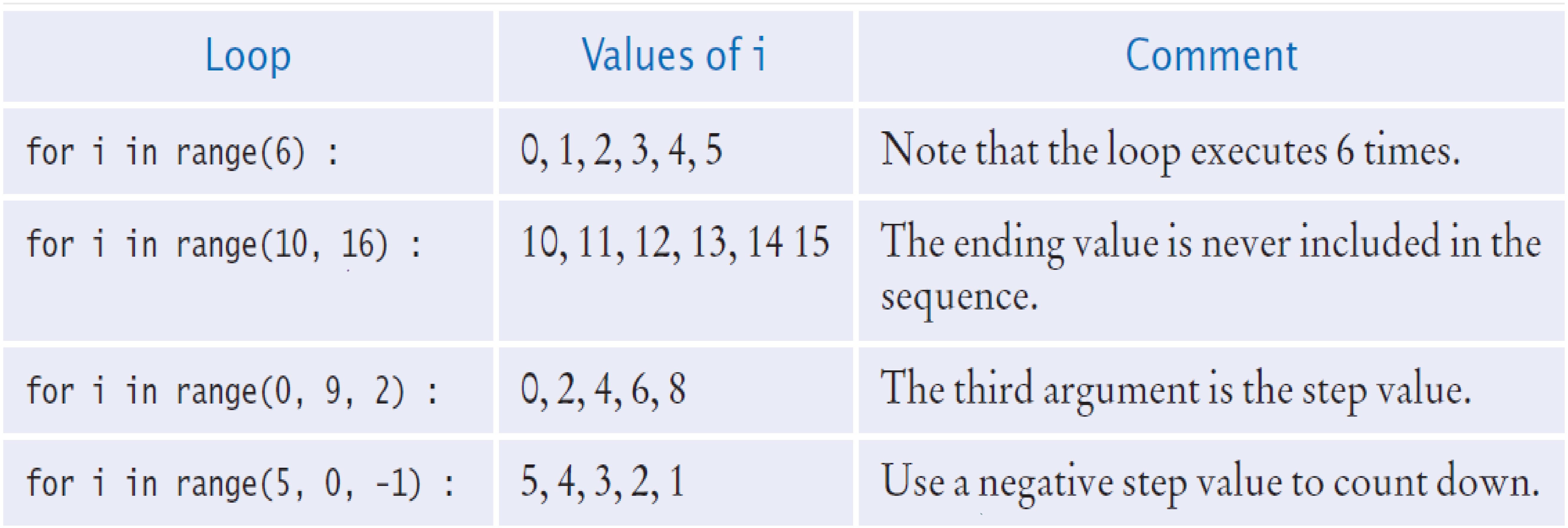
```
for i in range(1, 5) :
    print(i)   # Prints 1, 2, 3, 4
```

With two arguments, the sequence starts with the first argument.

```
for i in range(1, 11, 2) :
    print(i)   # Prints 1, 3, 5, 7, 9
```

With three arguments, the third argument is the step value.

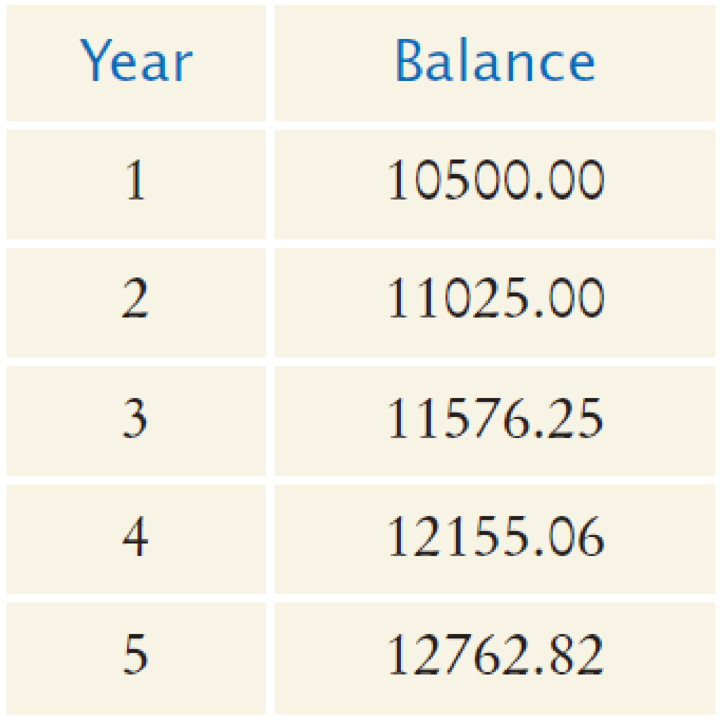
For Loop Examples



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.

For Loop

- **Example:** Now, let’s write a program to print the balance of our saving account over a period of years (entered by the user) with the initial balance of 10000 NOK and the rate of 5%.



Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

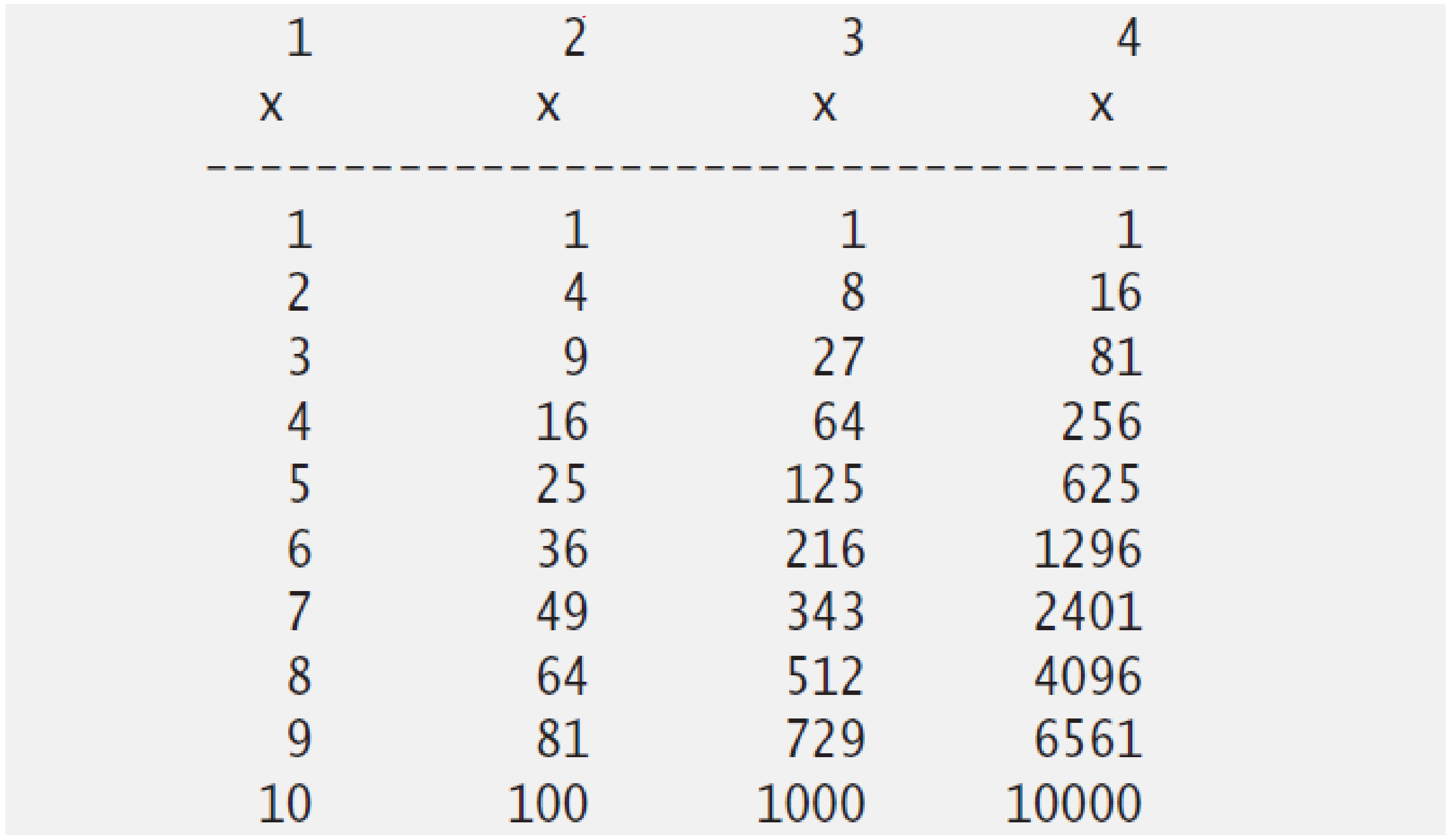
For Loop VS While Loop

- For loop
  - know number of iterations
  - can end early via break
  - uses a counter
  - can rewrite a for loop using a while loop

- While loop
  - unbounded number of iterations
  - can end early via break
  - can use a counter but must initialize before loop and increment it inside loop
  - may not be able to rewrite a while loop using a for loop

Nested Loops

- When the body of a loop contains another loop, the loops are **nested**. A typical use of nested loops is **printing a table** with rows and columns. For example, consider the following power table:



```
    1          2          3          4
   x          x          x          x
-----------------------------------
    1          1          1          1
    2          4          8         16
    3          9         27         81
    4         16         64        256
    5         25        125        625
    6         36        216       1296
    7         49        343       2401
    8         64        512       4096
    9         81        729       6561
   10        100       1000      10000
```

Nested Loop Examples

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

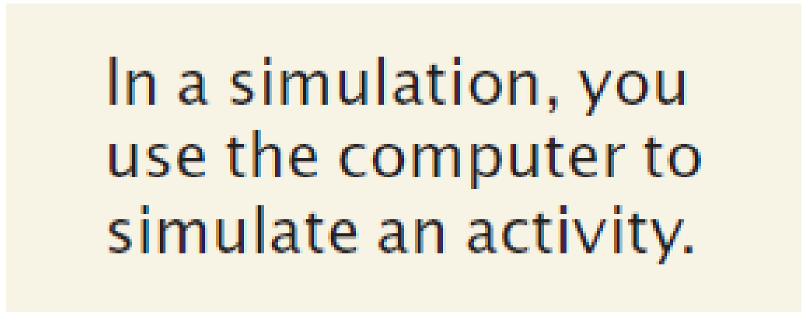
Nested Loop Examples

```
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.

Random Numbers and Simulations

- A **simulation program** uses the computer to simulate an activity in the real world (or an imaginary one).
- Simulations are commonly used for predicting climate change, analyzing traffic, picking stocks, and many other applications in science and business.
- In many simulations, one or more loops are used to modify the state of a system and observe the changes.


In a simulation, you use the computer to simulate an activity.

Random Numbers and Simulations

- Many events in the real world are difficult to predict with absolute precision, yet we can sometimes know **the average behavior** quite well. For example, a store may know from experience that a customer arrives every five minutes.

- The Python library has a random number generator that produces numbers that appear to be completely random.

- Calling **random()** yields a **random floating-point number** that is **≥ 0** and **< 1**. Call random() again, and you get a different number. The random function is defined in the random module.

Random Numbers and Simulations

- Generating a random integer within a given range:
- **randint (a, b)**
- The function is defined in the **random module**, returns a random integer that is between ***a*** and ***b***, **including the bounds themselves**.
- **Example:** Simulating Die Tosses

End of Chapter 4