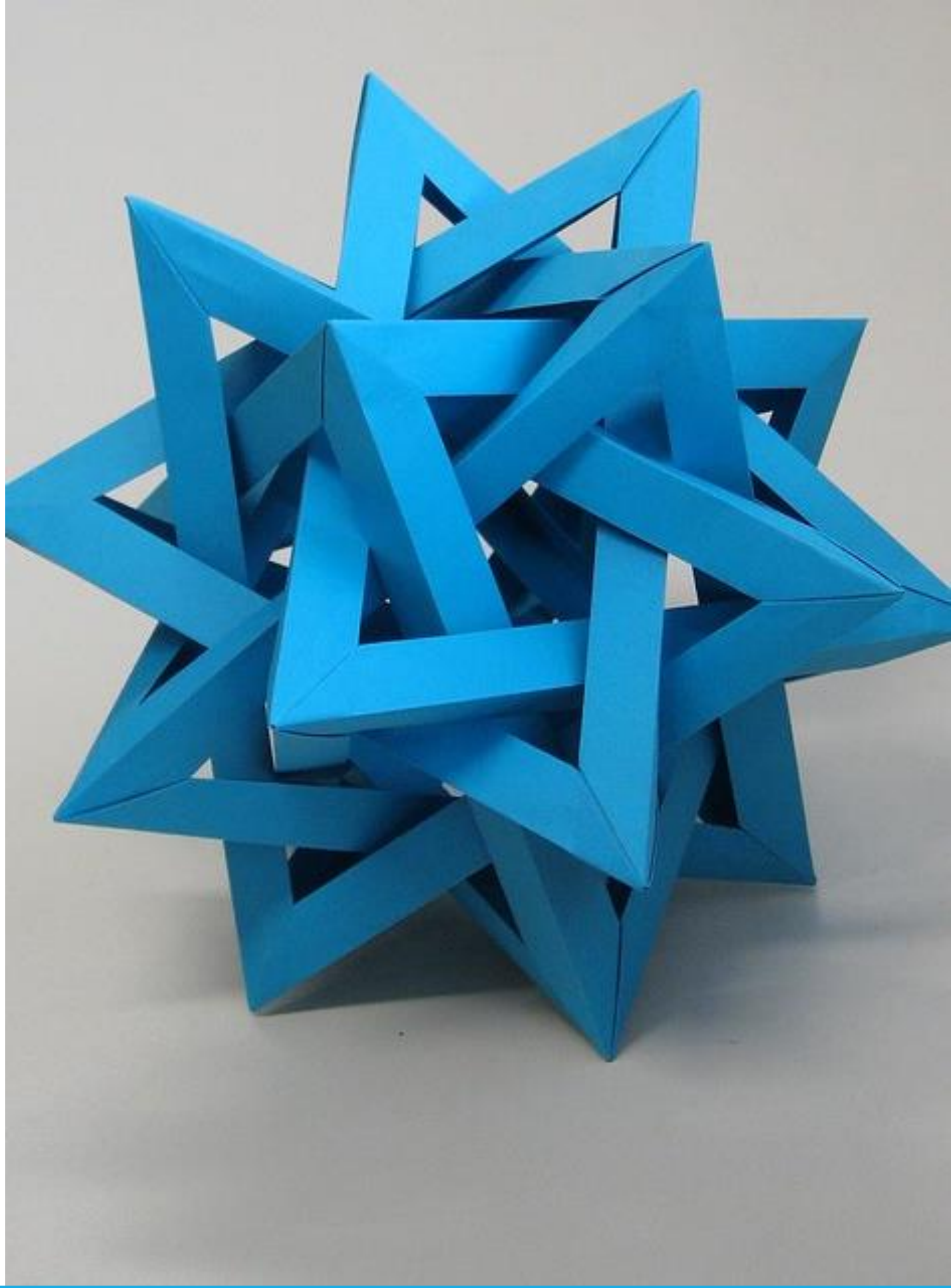
POLITECNICO DI TORINO
Dipartimento di Automatica e Informatica

Unit P4: Loops

LOOPS, REPETITIONS, CYCLES, RANGES, SEQUENCES

Chapter 4



Unit Goals

- To implement **while** and **for** loops
- 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

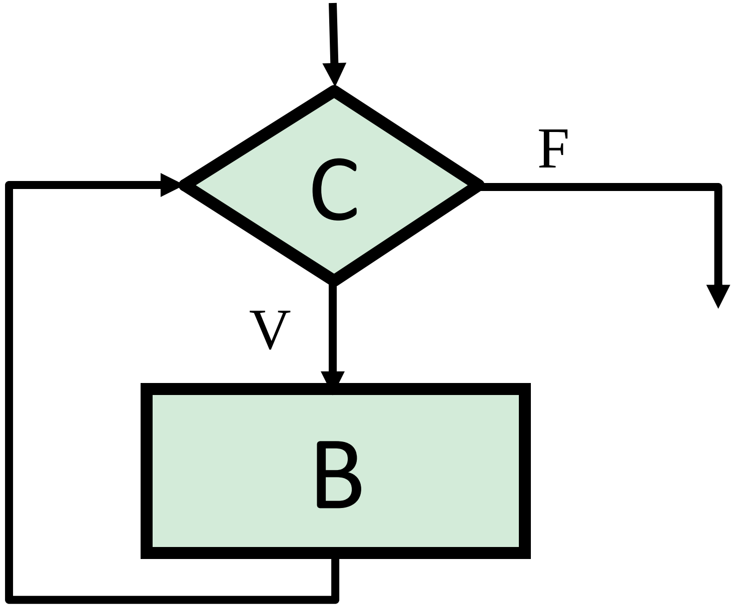
The `while` Loop

📘 4.1

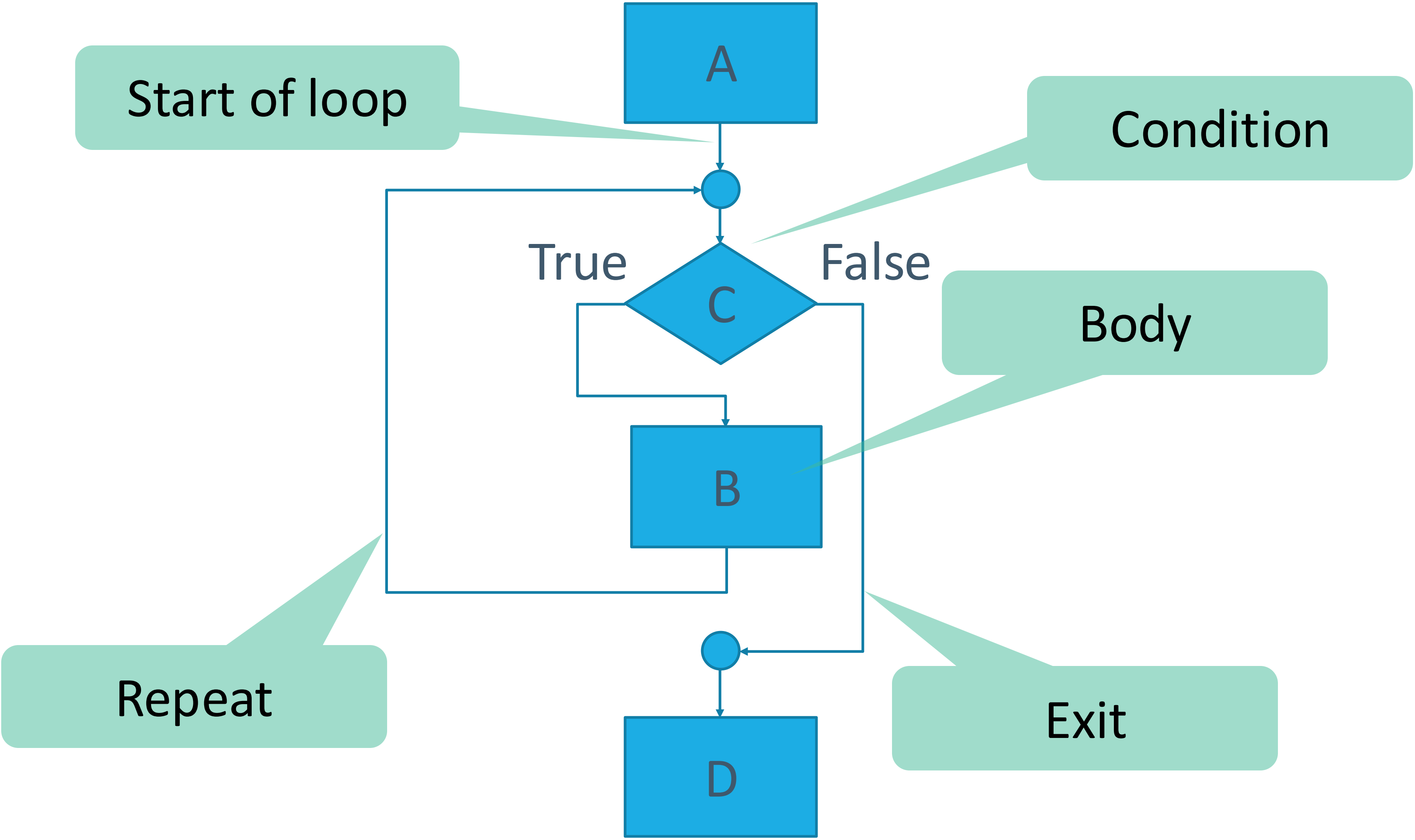
The `while` Loop

- Loops are used to repeat several times a block of instructions
  - With different values of the variables
  - Until some ‘exit condition’ becomes true

```
while C:
   B
```

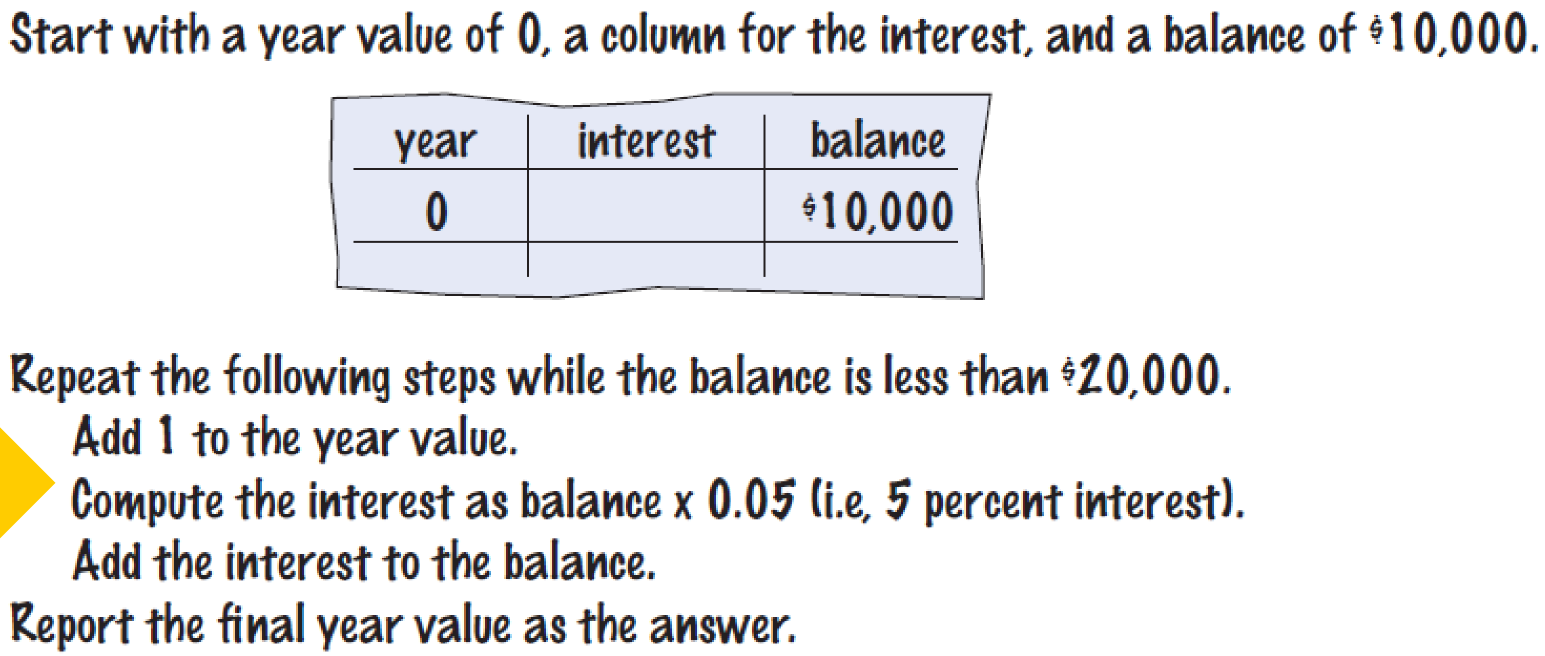


While loop - flowchart



Example

- Compound interest algorithm (Chapter 1)



Start with a year value of 0, a column for the interest, and a balance of $10,000.

year	interest	balance
0		$10,000

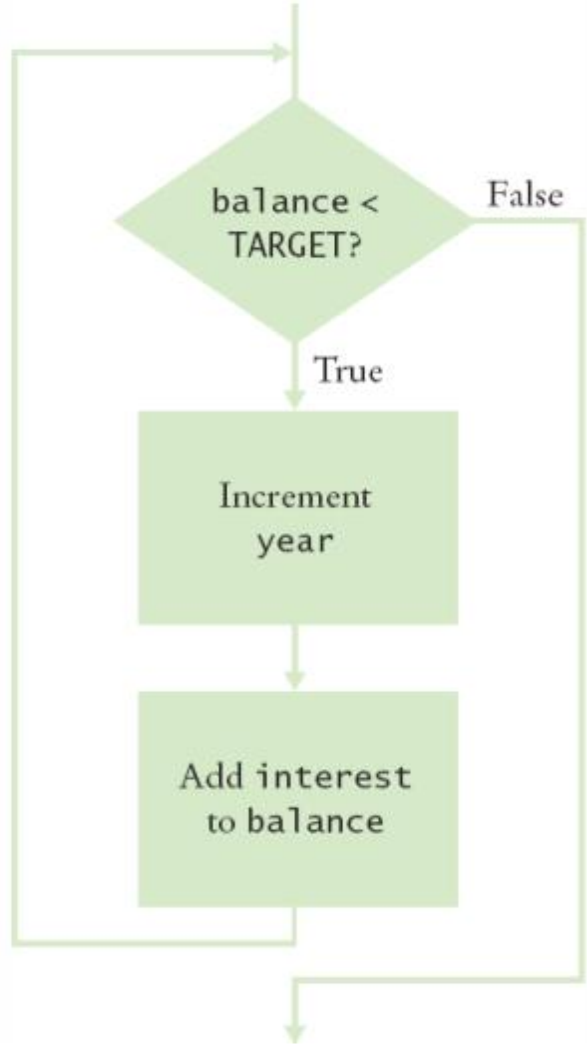
```
Repeat the following steps while the balance is less than $20,000.
    Add 1 to the year value.
    Compute the interest as balance x 0.05 (i.e, 5 percent interest).
    Add the interest to the balance.
Report the final year value as the answer.
```

Steps

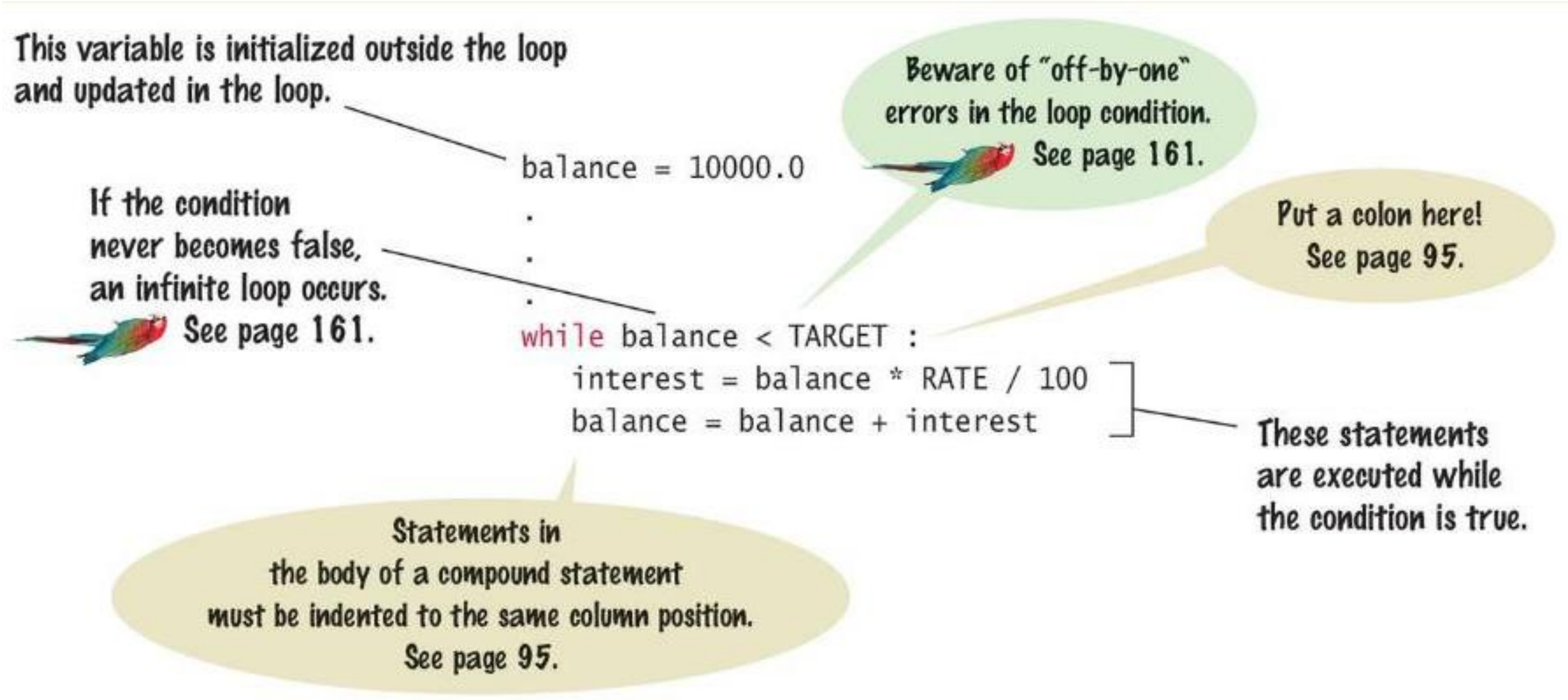
Example

```
balance = 10000.0
target = 20000.0
year = 0
RATE = 5.0
while balance < target :
    year = year + 1
    interest = balance * RATE/100
    balance = balance + interest
```

- A loop executes instructions repeatedly while its condition is True.



Syntax: `while` Statement



```
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 161.
Beware of "off-by-one" errors in the loop condition. See page 161.
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.

Count-Controlled Loops

- A **`while`** loop can be controlled by a **counter**
  - Counts the number of iterations already done
  - Stops when a predefined number of iterations has been completed
  - **In this case, we know in advance how many iterations we will do**

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

Event-Controlled Loops

- A `while` loop that is controlled by a condition that, sooner or later, will become False
  - **In this case, the number of iterations is not known in advance**

```
balance = input(“Initial balance:”)
while balance <= TARGET:    # Check the loop variable
    year = year + 1
    balance = balance * 2   # Update the loop variable
```

In this example, the number of iteration depends on balance (which is read from the user)

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
```

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

Nothing changes in the condition from one iteration to the next: Infinite loop!

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 am I done?
  - If you start at 0, use “<“
  - 0, 1, 2, 3, 4

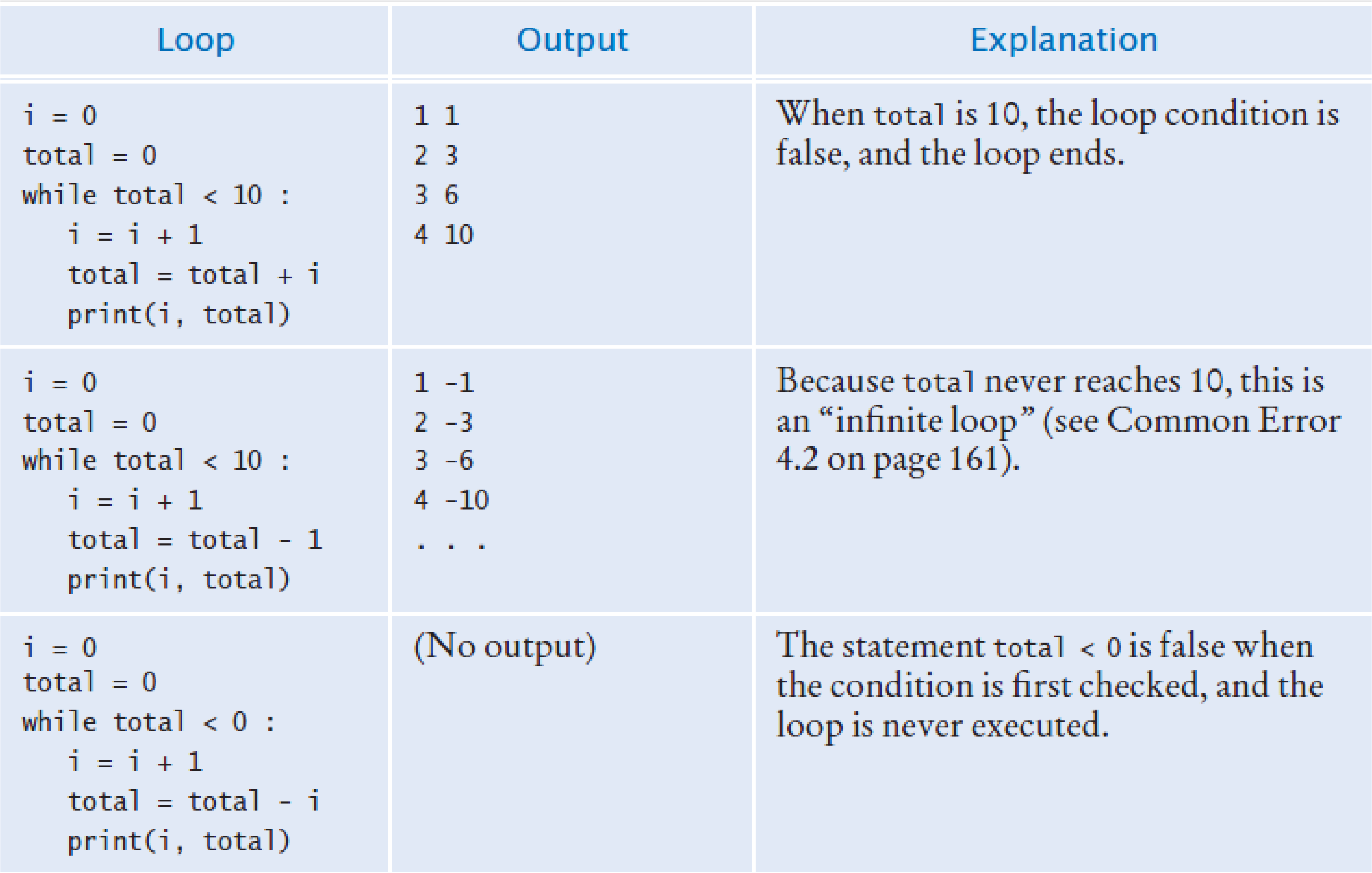
```
finger = 0
FINGERS = 5
while finger < FINGERS :
   # paint finger
   finger = finger + 1
```

If you start at 1, use “<=“

1, 2, 3, 4, 5

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

`while` Loop Examples



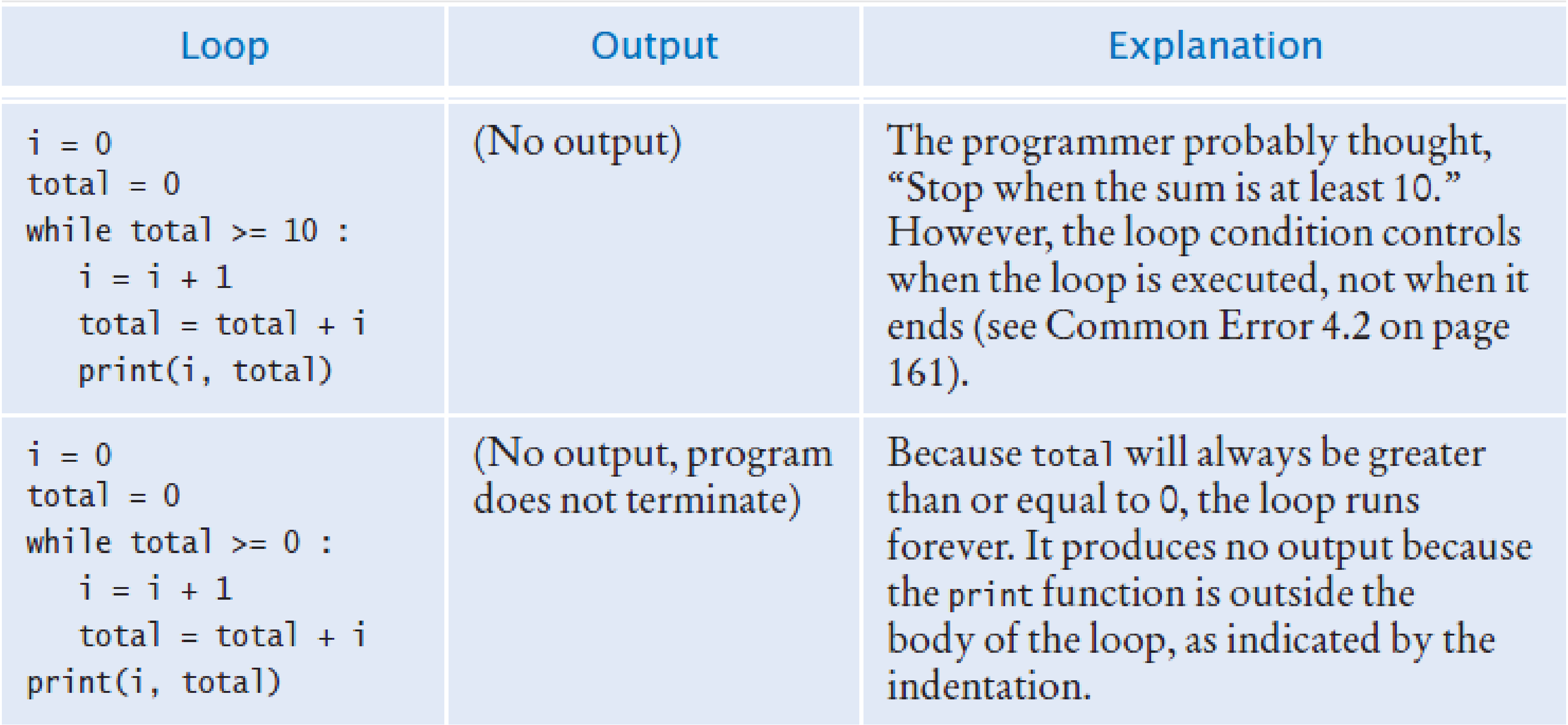
Loop	Output	Explanation

`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.

Mixed loops

- Not every loop is count-controlled or event-controlled
- Example: read a sequence of 10 numbers, but stop if you read 0

```
count = 0
ok = True
while (count < 10) and ok :
   a = int(input('Number: '))
   if a==0:
      ok = False
   print(f'Number {count+1}={a}')
   count = count + 1
```

OK is called a “Flag” variable

Summary of the `while` Loop

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

Sentinel Values

📘 4.3

Working with Sentinel Values

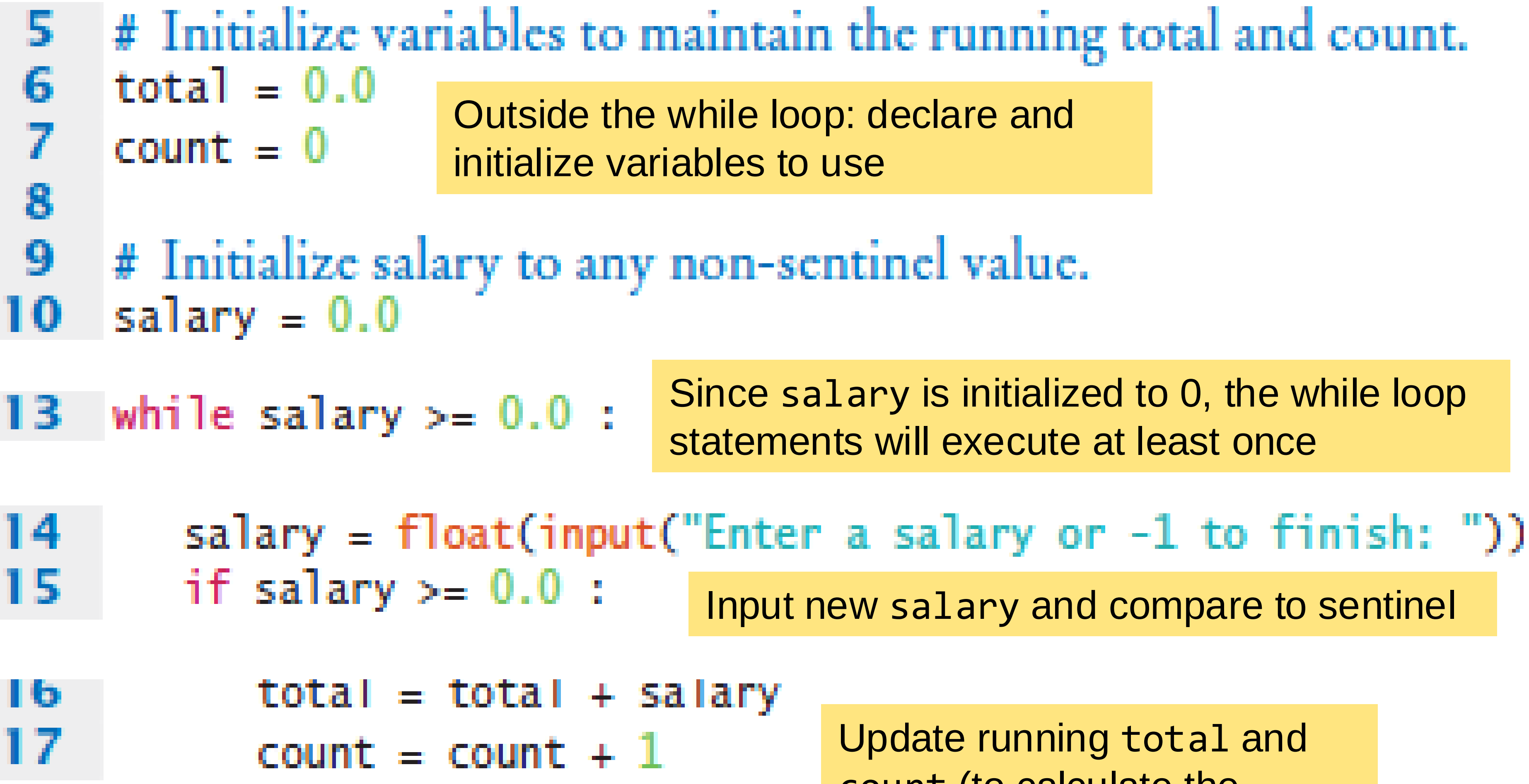
- Whenever you read a sequence of inputs, you need to have some method of indicating the end of the sequence
- When you don’t know how many items are in a sequence, you can use a ‘special’ value to signal the “last” item
  - This special value is called “sentinel”
  - A sentinel value denotes the end of the data, but it is **not** part of the data.

- Examples:
  - **Use -1 to indicate the end of a sequence of positive numbers.**
  - Use ’.’ to indicate the end of a sequence of letters.

- Not always possible to identify a sentinel
  - Example: program that works on any float number…

Sentinel.py (1)

The `if` checks to make sure we are not processing the sentinel value



```
# Initialize variables to maintain the running total and count.
total = 0.0
count = 0

# Initialize salary to any non-sentinel value.
salary = 0.0

while salary >= 0.0 :

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

        total = total + salary
        count = count + 1
```

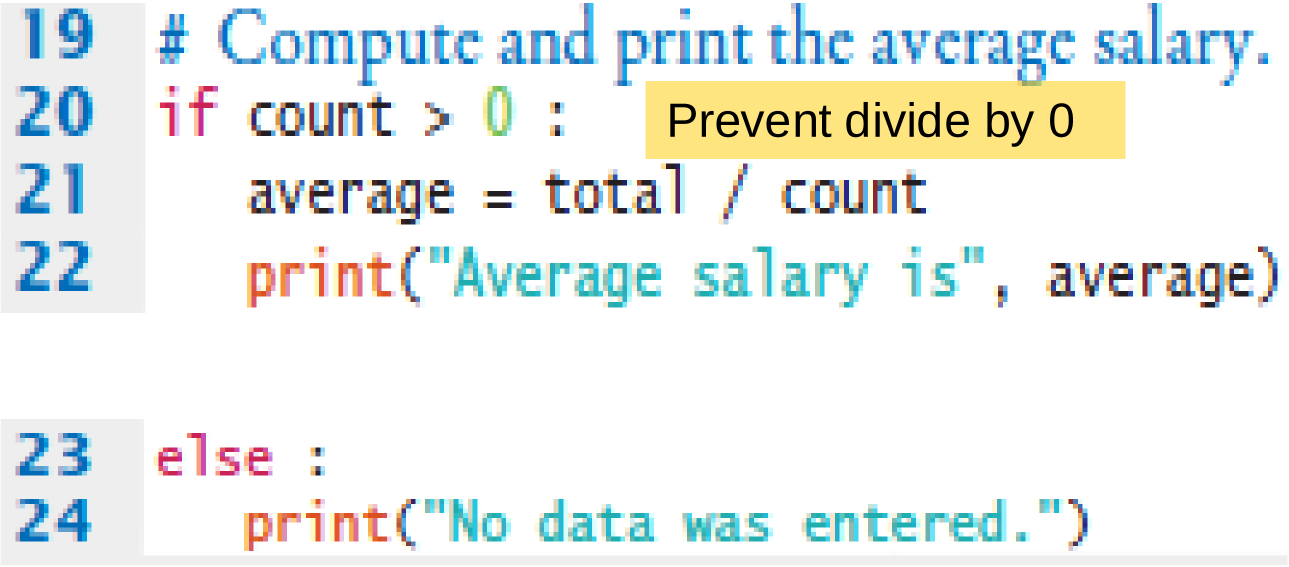
Outside the while loop: declare and initialize variables to use

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)

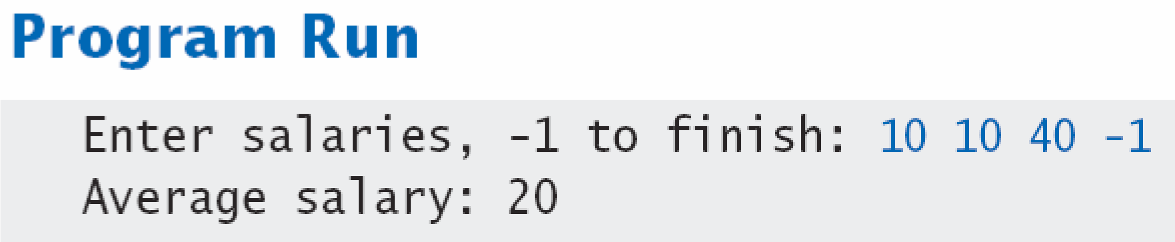
Sentinel.py (2)



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

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

Prevent divide by 0

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



Program Run

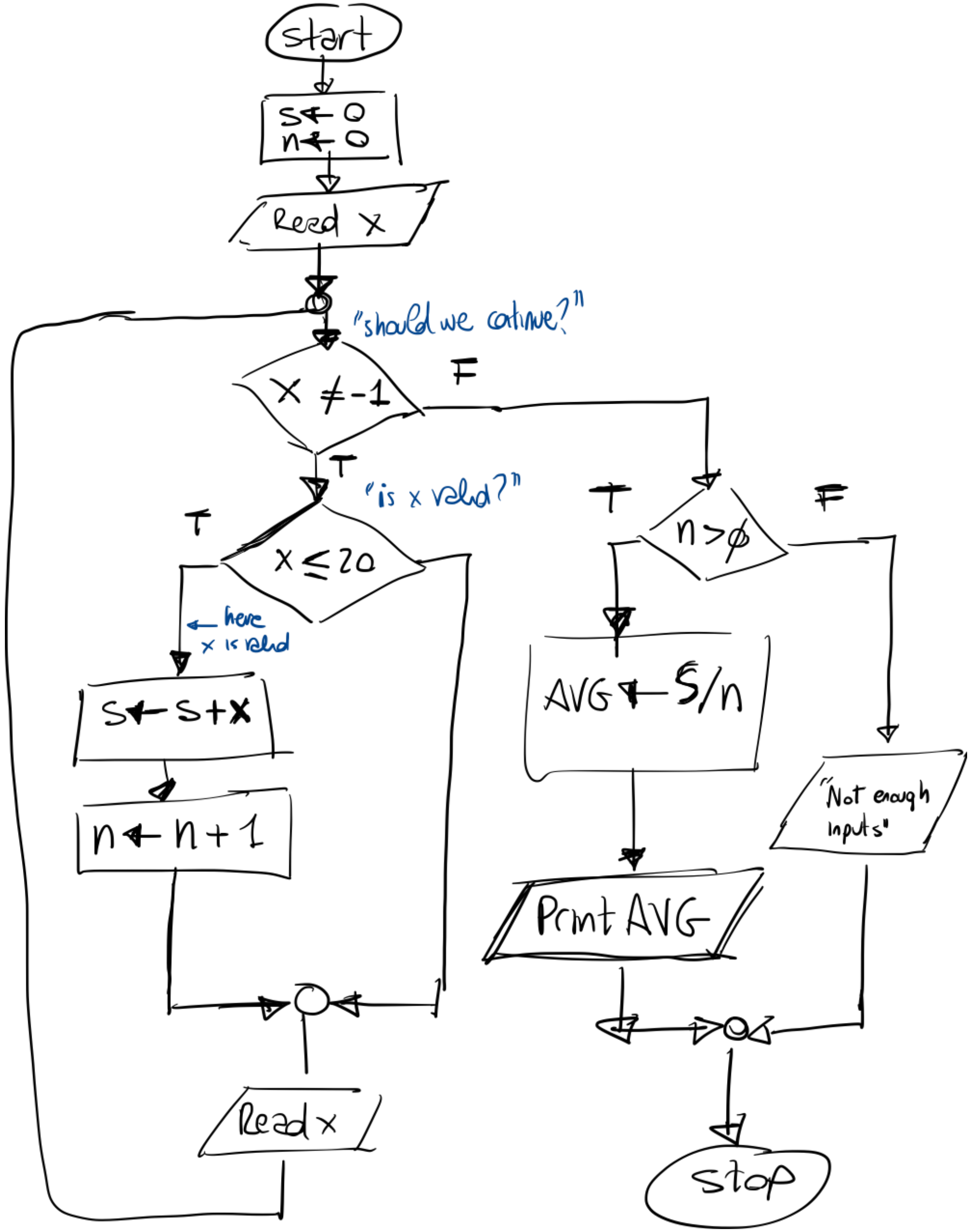
```
Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20
```

Another Example

- *Enter a series of values (positive integers) from the keyboard.*
  The series ends when you enter the value -1.
  Calculate and print the average ignoring the >20 values.

- **Seen in the video-lecture on Flowcharts!**

Another Example



Another Example

- Although there are multiple schemes to work with sentinels, one that works well is the following:
  - Read input once **before the loop** to initialize the checked variable (x in this case)
  - Read input again **inside the loop body** (usually at the end) to update the variable
- The previous salary example uses a different scheme:
  - Initialize salary to a value != from the sentinel to enter the loop, then immediately read input

```
x = int(input("Inserisci un valore o -1 per finire: "))
while x != -1:
    if x <= 20:
        S = S + x
        n = n + 1
    x = int(input("Inserisci un valore o -1 per finire: "))
```

Empty String as Sentinel

- The empty string can be used as a sentinel:
- The input() function always returns a string, which will be empty if the user presses Enter without typing anything

General Scheme

```
s = input(“Insert value: ")
while s != "" :
   convert the value if needed
   process the value
   s = input(“Insert value: ")
```

Example: accumulate values in total

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

Boolean Variables and Sentinels

- A boolean variable can be used to control a loop
  - In this case, it is typically 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

📘 4.5

Common Loop Algorithms

- Sum and Average Value
- Counting elements matching a certain condition
- **Asking input until a valid value is provided**
- **Maximum and Minimum**
- **Comparing Adjacent Values**

Average of ‘count’ values

- Compute the total sum of the values
- Initialize count to 0
  - Increment per each input
- Check for count > 0
  - Before dividing!

```
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
```

Counting Matches *(e.g., Negative Numbers)*

- Counting Matches
  - Initialize negatives to 0
  - Use a while loop with a sentinel
  - Add to negatives if the condition matches

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

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

Ask Input Until Valid

- Initialize a Boolean sentinel (flag) to **False**
- Test the flag in the 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 provided inputs

Ask Input Until Valid

- In general, calling **C** the condition that the input must respect

```
valid = False
while not valid :
   value = (input(“? "))
   if C:
      valid = True
   else :
      print("Invalid input.")
```

Or, without flag

```
value = input(“? ")
while not C:
   print("Invalid input.")
   value = input(“? ")
```

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
  - Update largest if necessary

Why can't we just use `max()`?

```
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: ")
```

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
  - Update smallest if necessary

Why can’t we just use `min()`?

```
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

- Many times, we need to operate on consecutive values in a series

- For example:
  - Verify if they are duplicate
  - Combine them, etc.

- This requires a slight change in our code pattern for reading a series of inputs.

Comparing Adjacent Values

- Example: check if the user inserts **two consecutive identical inputs**
- Get first input value
- Use **while** with sentinel as before… but this time we need to **”keep in memory” 2 values** at all times, so:
  - ○ Copy old `value` to `previous` variable
  - ○ Get next input into `value`
  - ○ Compare `value` and `previous`, and print if they are the same

```
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: ")
```

The `for` Loop

4.6

The `for…in` Loop

- Uses of a **`for…in`** loop:
  - The for loop can be used to iterate over the contents of any container

- A container is an object (like a string) that contains or stores a collection of elements → **Elements Provider**

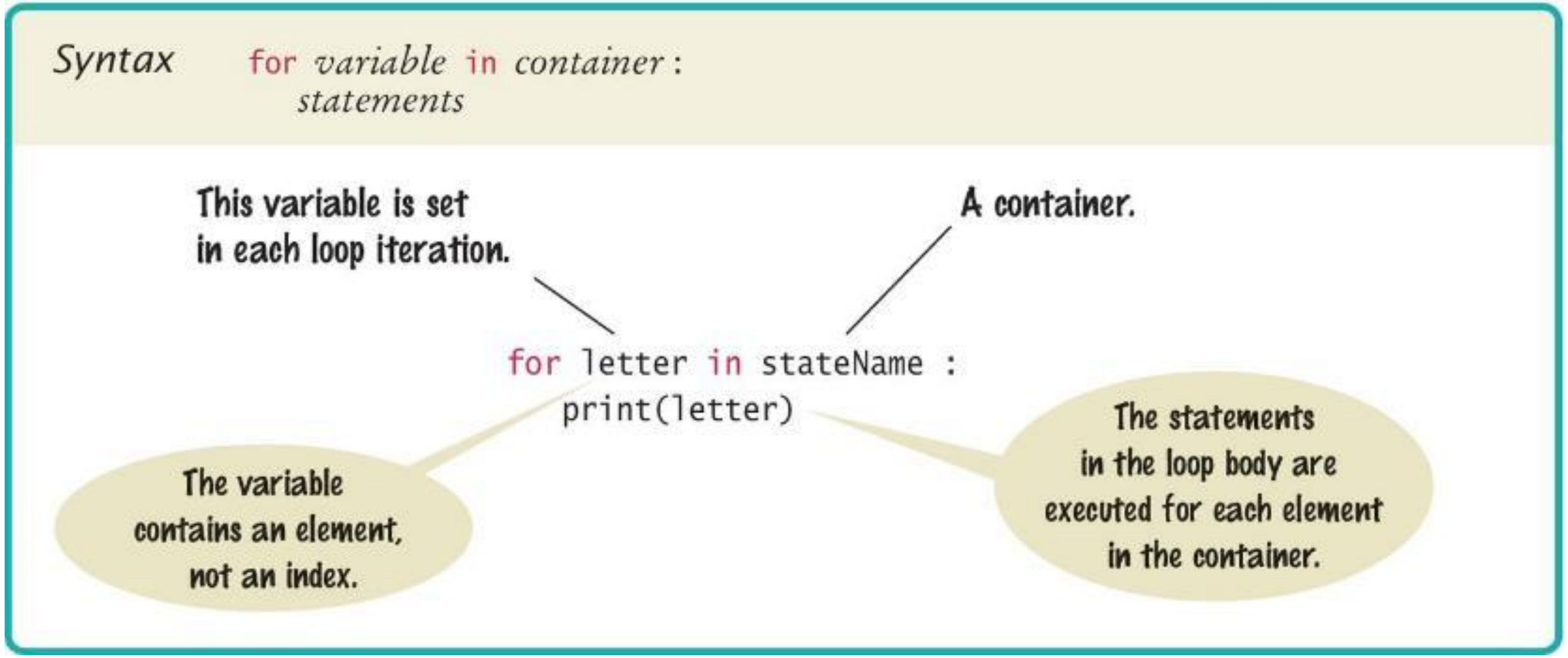
- Python has several types of containers
  - A string is a container that stores the collection of characters in the string
  - A list (Chapter 6) is a container of arbitrary values
  - A file (Chapter 7) is a container or rows of text

Basic loop

-
  ```
  for variable in elements provider:

        do something with variable
  ```

Syntax of a for Statement (Container)

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



Syntax

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

This variable is set in each loop iteration.

A container.

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

The variable contains an element, not an index.

The statements in the loop body are executed for each element in the container.

An Example of a for Loop

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

while version

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

for version

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 **letter** variable is assigned the sequence elements:
  - stateName[0], stateName[1], and so on.

In the example: “Virginia” is a container of the sequence “V”, “i”, “r”, “g”, “i”, “n”, “i”, “a”

The ‘range’ container

- A special container is a sequence of consecutive numbers
- Generated with the `range(N)` function
  - `range(N)` creates a sequence of integers from `0` to `N-1`
- Therefore, a `for` loop with a `range()` will execute as a **counter-based** `while` loop

The for Loop (2)

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

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

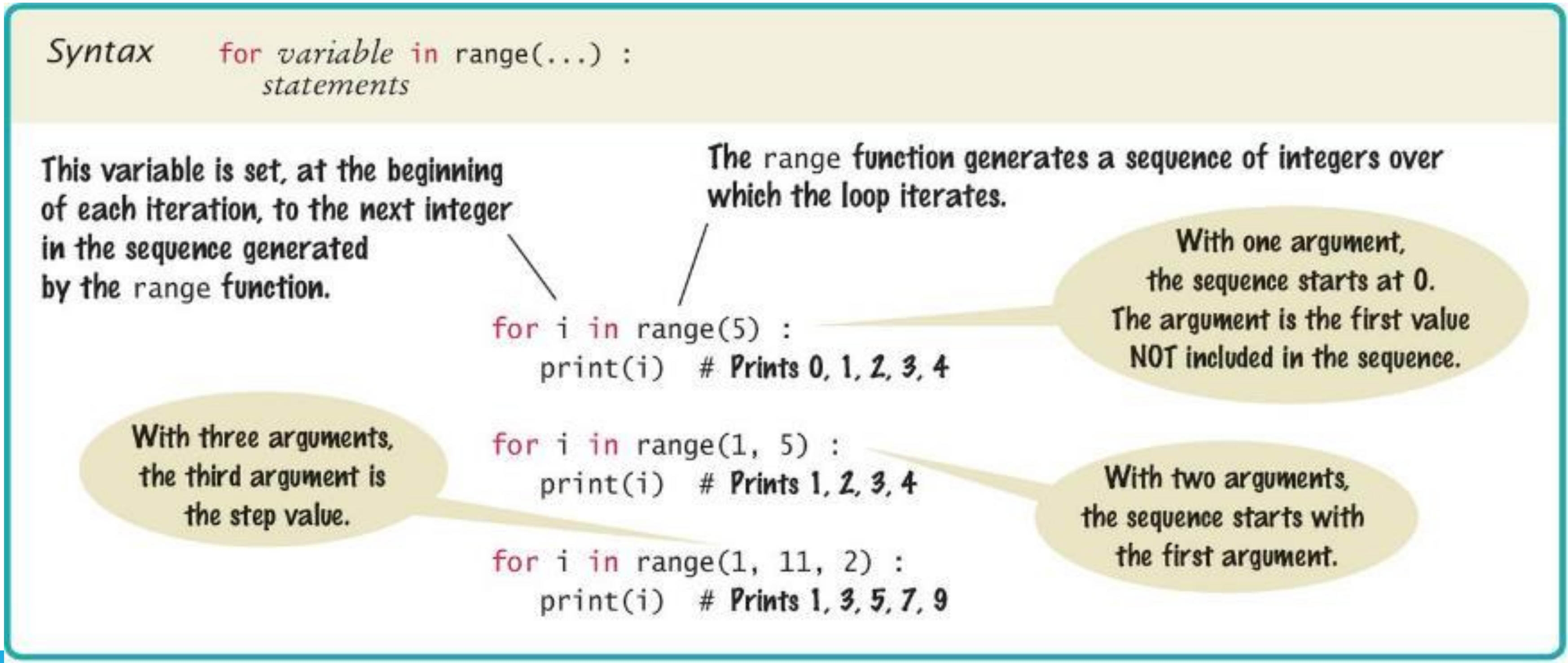
while version

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

for version

Syntax of a for Statement (Range)

- A for loop can be used as a **count-controlled loop** that iterates over a range of integer values.
- The `range` function generates a sequence of integers that can be used with the 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.

Good Examples of for Loops

- Keep the the loops simple!

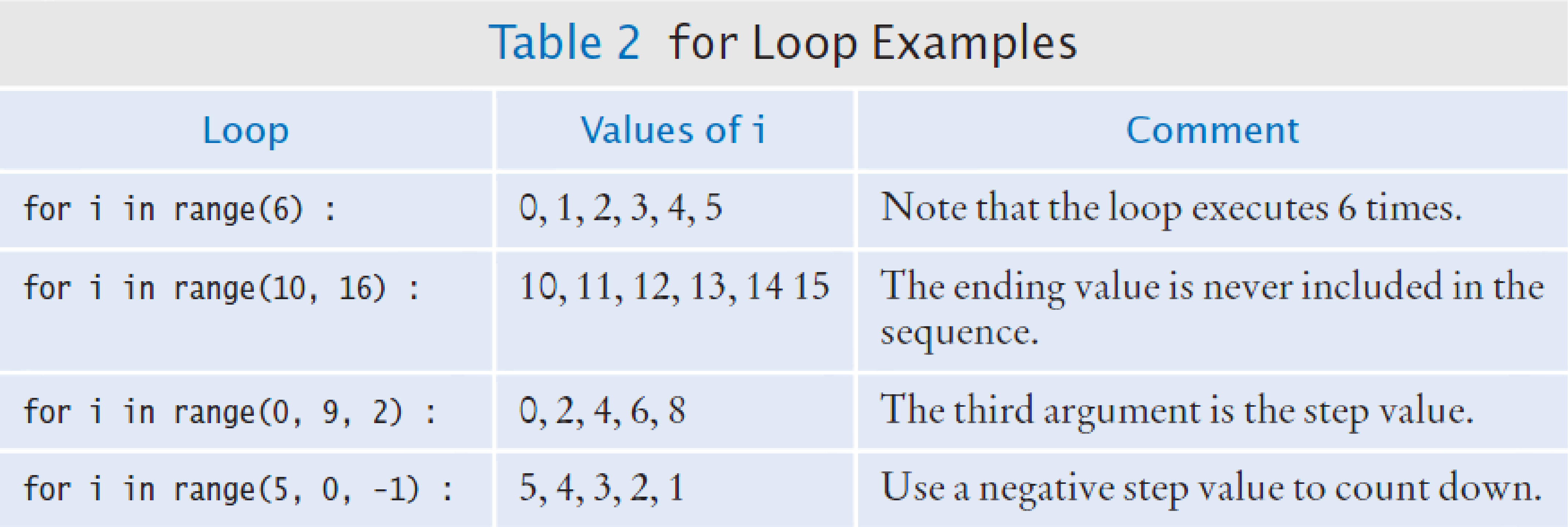


Table 2 for Loop Examples

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.

Cycling by index and value

Scanning strings with **for**

- When we need to examine each character of a string, **independently of its position**, we can use a simple **for.. in**:

```
for ch in string :
```

  - At each iteration, **ch** will contain the next character of the string

- If we need to operate on the **position** of some characters of the string, then we need to iterate using **range:**

```
for i in range(len(string)) :
```

  - At each iteration, **i** will contain the **index** of the characters in the string, from 0 to len(string) – 1.

Printing the characters and index in a string

WHILE -> cycling by index

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

The index needs to be update manually

FOR -> cycling by value

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

Very compact but the index is not available

```
i = 0
for letter in name :
    print(i, letter)
    i = i + 1
```

Very similar to the while solution

```
for i in range(len(name)):
    print(i, name[i])
```

Compact alternative

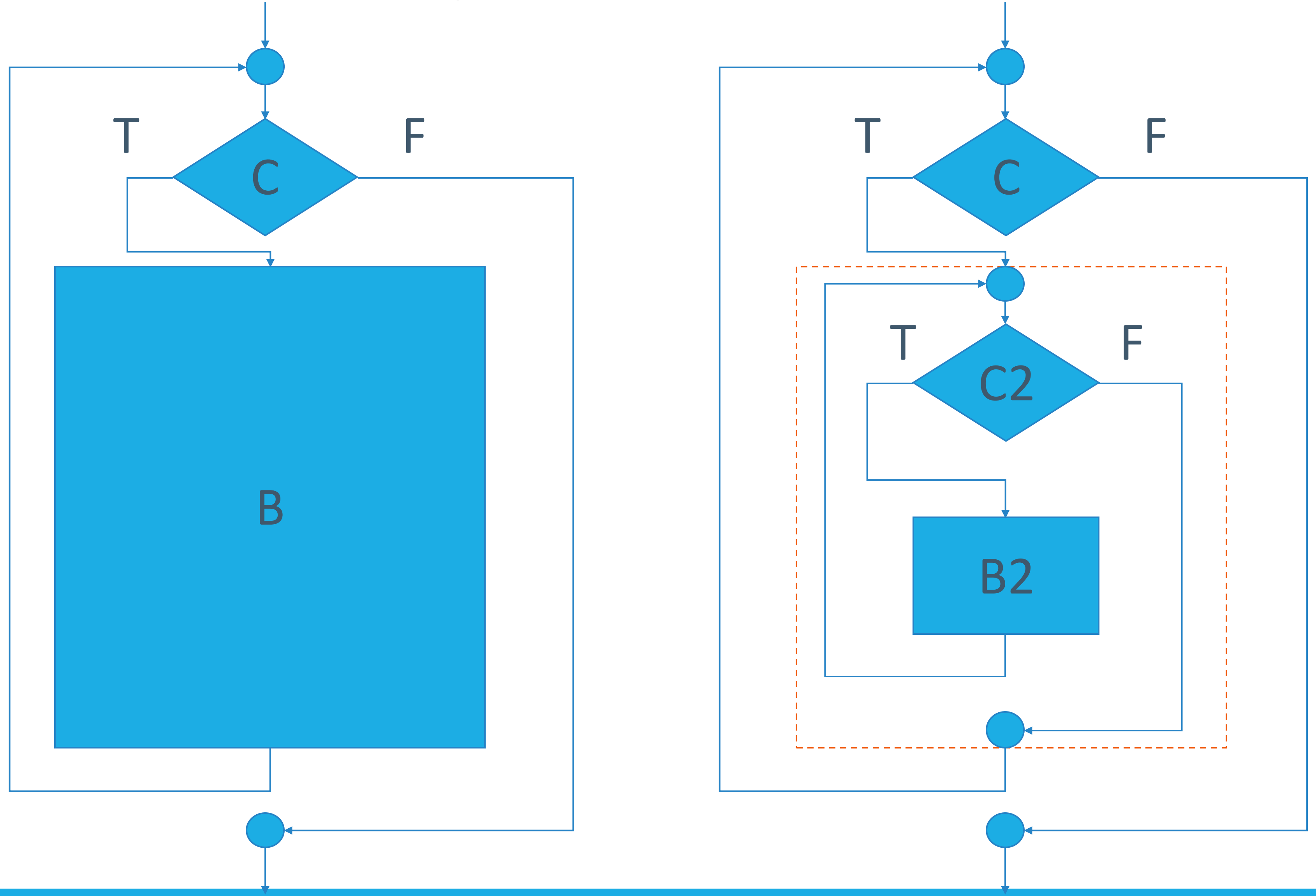
Nested Loops

📘 4.7

Loops Inside of Loops

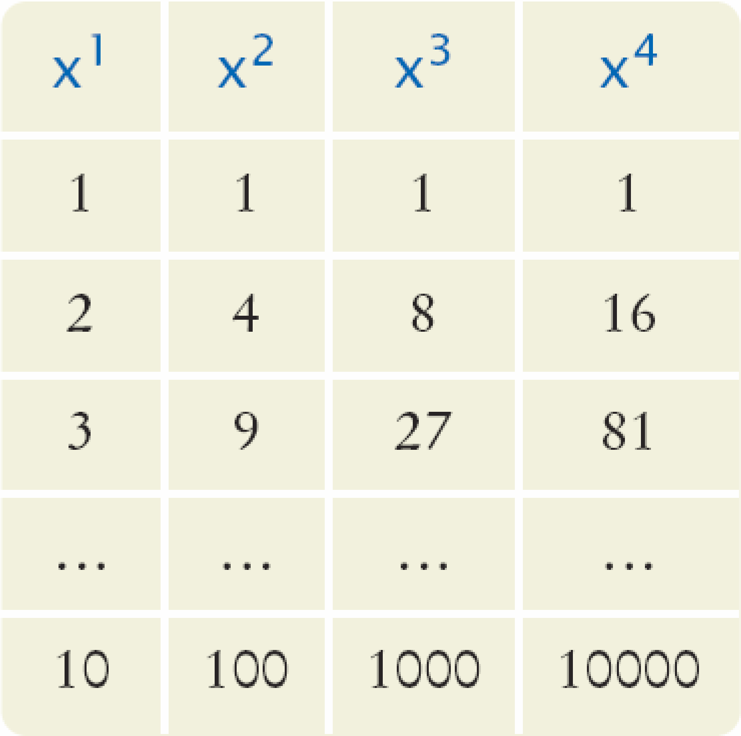
- We know how to nest `if` statements for making complex decisions
  - Remember: nesting requires to indent the code block
- Complex problems sometimes require **loops nested inside other loops**
- When?
  - **Multi-dimensional problems**
    - Example: operate on a “table”, or matrix
    - One loop iterates over the rows, and for each row, another loop iterates over the columns
  - Iterative problems that must be repeated multiple times

Nested while loops



Our Example Problem Statement

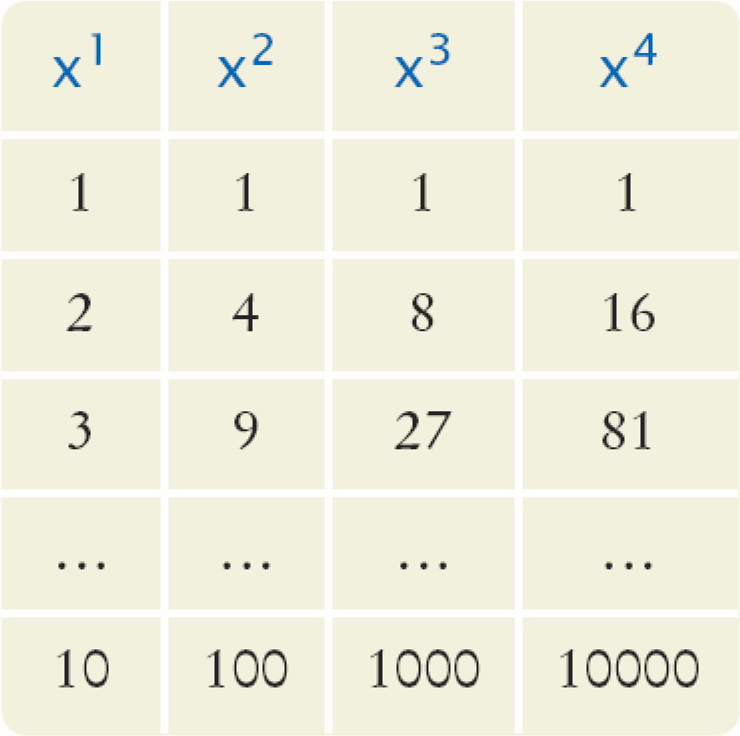
- Print a Table Header that contains x^1, x^2, x^3, and x^4
- Print a Table with four columns and ten rows that contains the powers of x^1, x^2, x^3, and x^4 with x ranging from 1 to 10



x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
...	...	...	...
10	100	1000	10000

Applying Nested Loops

- How would you print a table with rows and columns?
  - Print top line (header)
    - Use a for loop
  - Print table body…
    - How many rows are in the table?
    - How many columns in the table?
  - **Loop per row**
    - **Loop per column**
- In our example there are:
  - Four columns in the table
  - Ten rows in the table



x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

Pseudocode to Print the Table

- Print the table header

```
for x from 1 to 10
    print a new table row
    print a new line
```

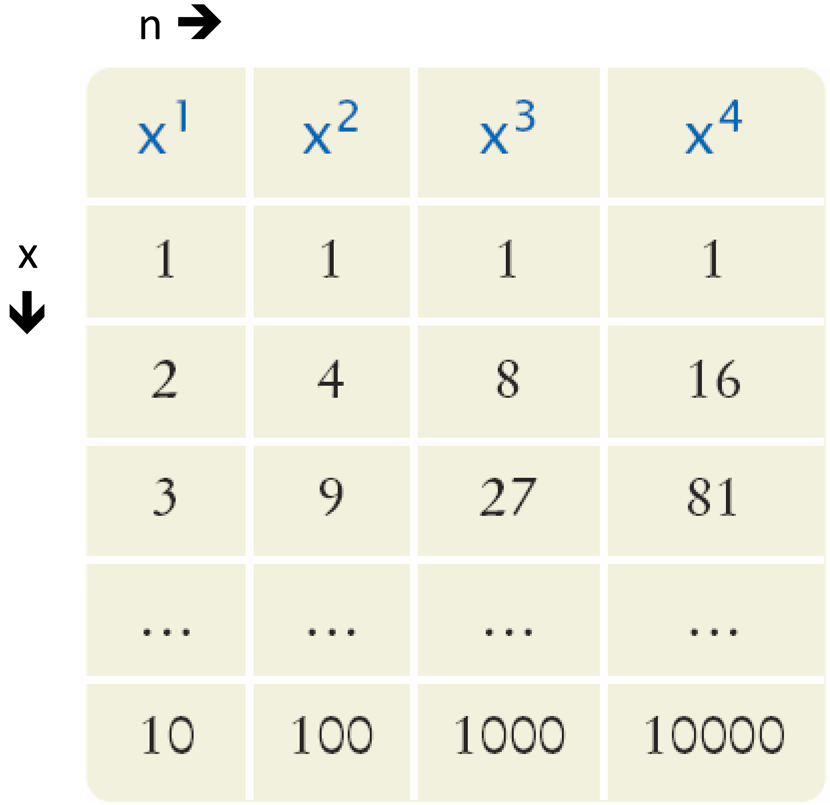
- How do we print a table row?

```
for n from 1 to 4
    print $x^n$
```

- We have to place this loop **inside the previous loop**
  - The inner loop is “**nested**” inside the outer loop

Pseudocode to Print the Table

```
print the table header
for x from 1 to 10
    for n from 1 to 4
        print $x^n$
    print a new line
```

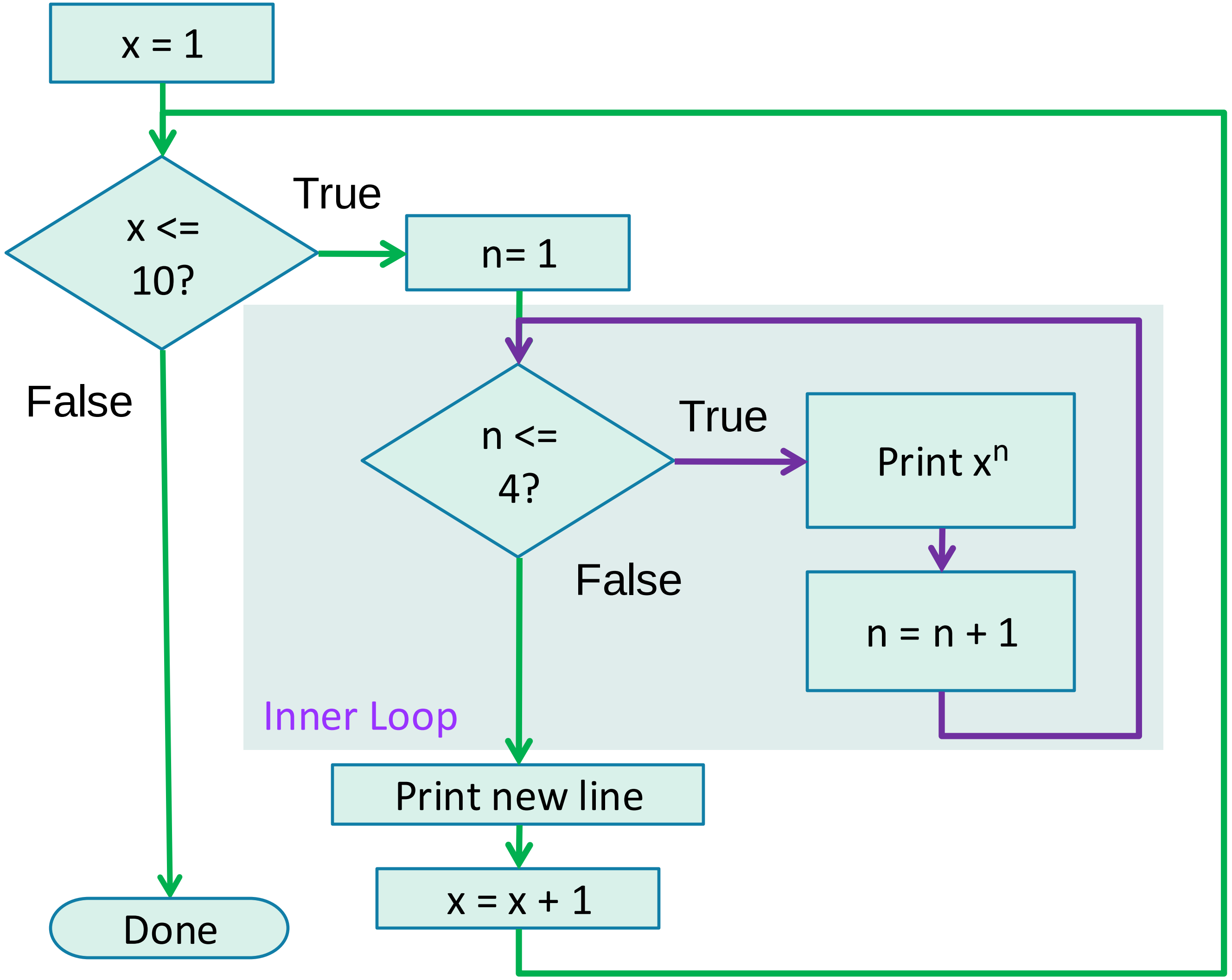


n →

x ↓

x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

Flowchart of a Nested Loop



Multiple print() with output on the same line

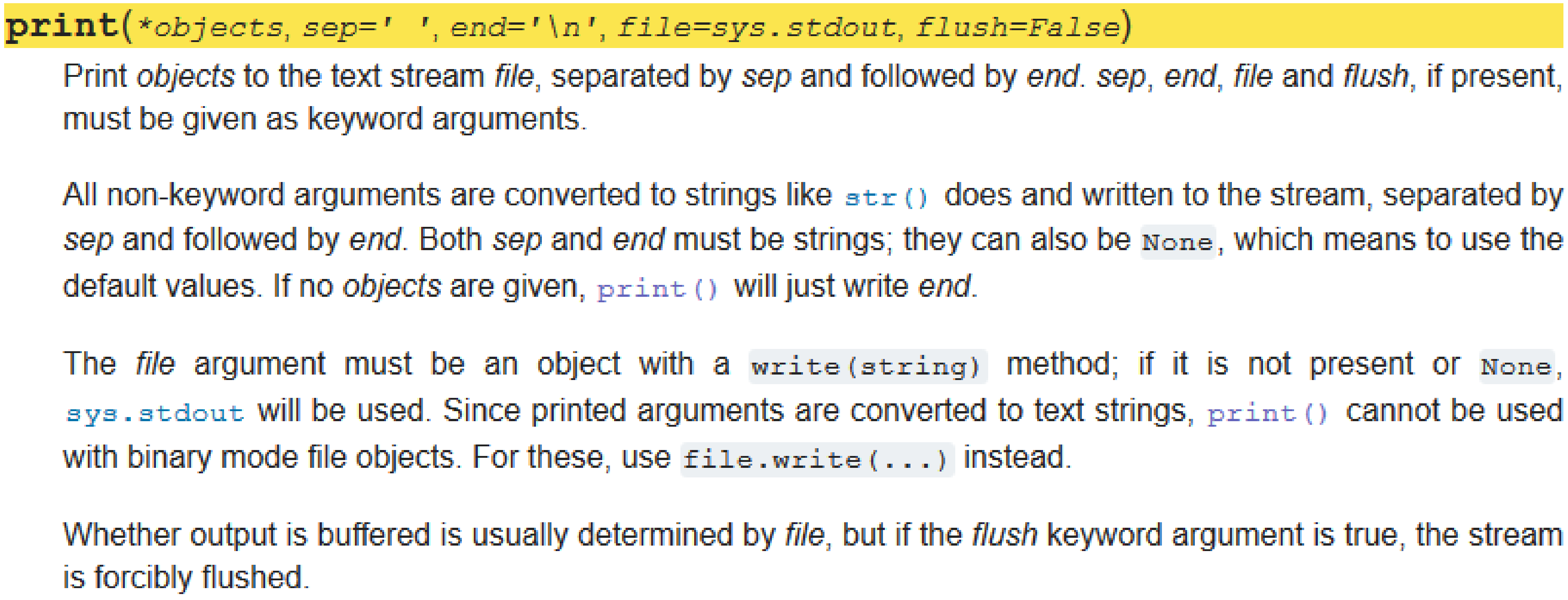
- By default, **print()** adds a newline at the end of the output.
- This can be changed using the ‘end’ optional parameter
  - Using **end=""** as the **last argument** to the `print` function prints an empty string *after* the arguments, instead of a new line
  - So, the output of the next `print` function starts on the same line
- For example, the two statements:

```
print("00", end="")
print(3 + 4)
```

Produce the output:

```
007
```

Optional parameters of the `print` function



`print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

- **sep=**
  - Defines the separator that will be inserted if you print several valued
  - Default: space
  - `print(hour, min, sec, sep=':')`

- **end=**
  - Defines what to print at the end of each line
  - Default: newline
  - `print('Hello', end='')`

Exercise

- Take as input a number N > 0
- Draw a ‘right triangle’ of side N using asterisks

  - Es: N=4

```
*
**
***
****
```

  - Es: N=7

```
*
**
***
****
*****
******
*******
```

Exercise

- Reasoning: similar to the table
    - There are N rows => for loop
    - What's in the **i-th row**?
        - Let's make a table:

Riga	Output
1	*
2	**
…	
…	
i	

Exercise

- Reasoning: similar to the table
  - There are N rows => for loop
  - What's in the **i-th row**?
    - Let's make a table:

Riga
1
2
…
…
i

What if I wanted my triangle to look like this?

Ex: N=3

```
  *
 ***
*****
```

Exercise (Variant 2)

- Hint: we must add spaces before asterisks. How many in the **i-th** row?

```
  *
 ***
*****
```

N=3

```
   *
  ***
 *****
*******
```

N=4

Exercise (Variant 3)

- What if we want to draw this figure?

```
   *          N=3
  * *
 *****

   *          N=4
  * *
 *   *
*******
```

Hint: the last row has to be handled separately

Nested Loop Examples

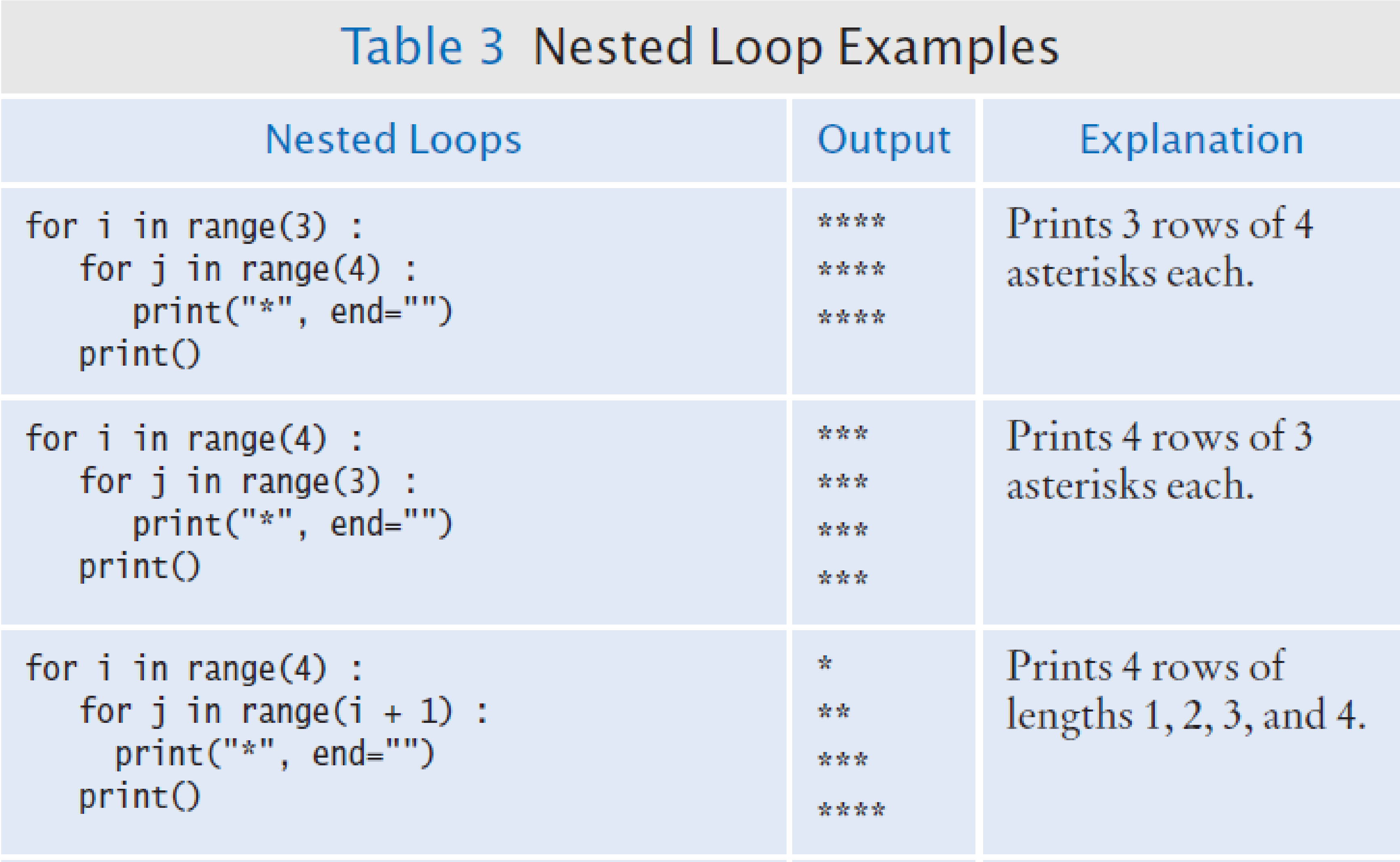


Table 3 Nested Loop Examples

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.

Exercise: Repeated Iterative Procedure

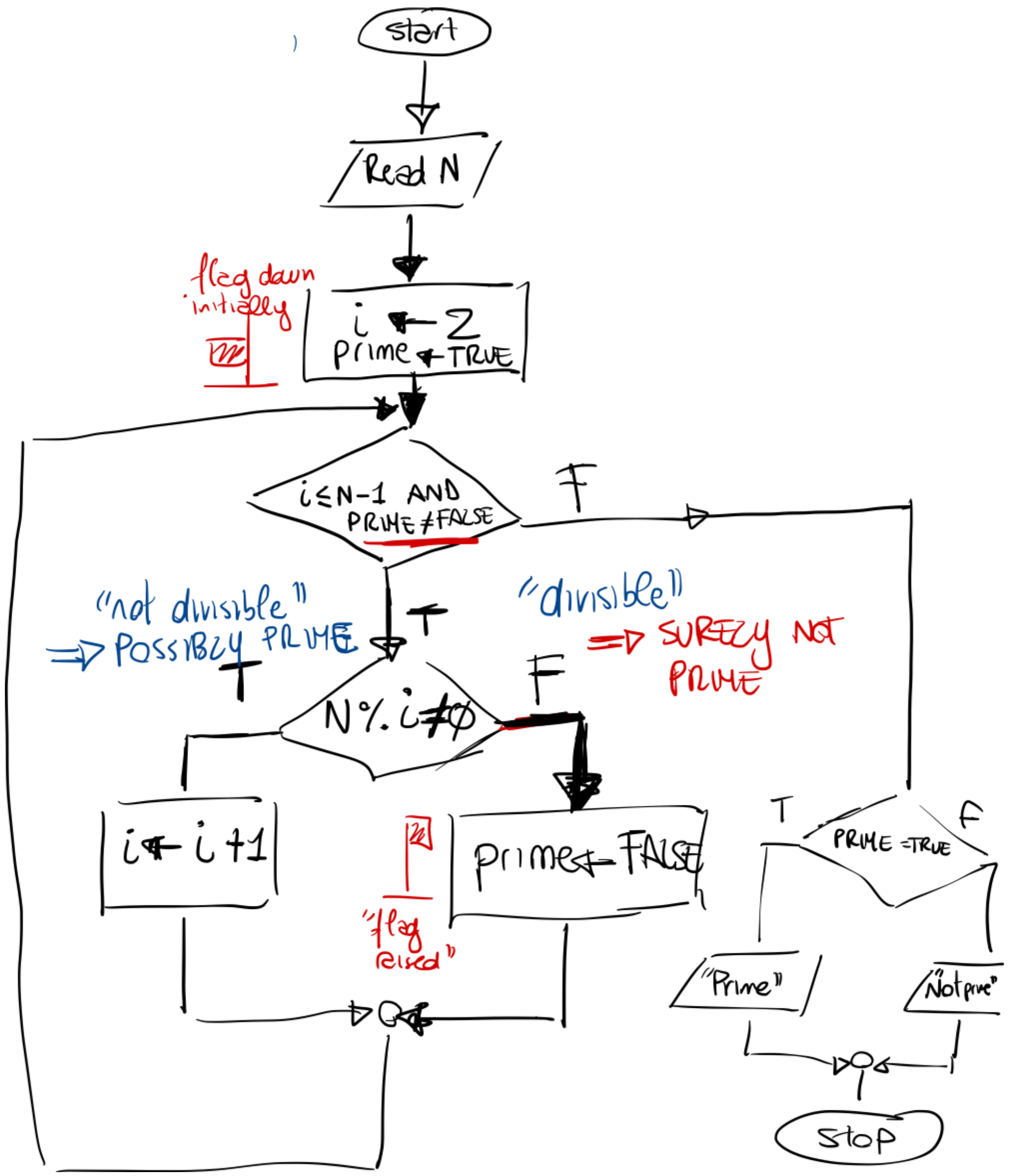
📘 4.8

Exercise

- Another typical use of **nested loops** is when you need to repeat an **iterative procedure** multiple times.
- Example: take as input a series of values, and check if each of them is prime
- Structure:

```
Foreach number n in the series:
 Check if n is prime (see flowchart of video-lecture)
```

Exercise


start
Read N
flag down initially
i ← 2
prime ← TRUE
i≤N-1 AND PRIME≠FALSE
F
T
"not divisible"
⇒ POSSIBLY PRIME
"divisible"
⇒ SURELY NOT PRIME
T
F
N%i≠0
i ← i+1
prime ← FALSE
"flag raised"
T
F
PRIME = TRUE
"Prime"
"Not prime"
stop

Processing Strings

📘 4.8

Processing Strings

- A common use of loops is to process Strings.

- For example, you may need to:
  - Count the number of occurrences of one or more characters
  - Verify that the contents of a string meet certain criteria.

String Processing Examples

- **Counting Matches**
- Finding All Matches
- Finding the First or Last Match
- **Validating a String**
- Building a New String

Counting Matches

- Example: *count the number of uppercase letters* contained in a string.

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

Counting Vowels

- Suppose you need to count the vowels within a string
- We can use a `for` loop to check if each character in the is contained in the **string of vowels** “`aeiuo`”
- We consider the character after converting it to **lowercase** so that we can count both ‘a’ and ‘A’ as vowels (etc)

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

Finding All Matches

- When we 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 iterate over the positions (using **range()**):
  - Test the character at each position
  - If uppercase, print the position **i** in the string.

```
sentence = input("Enter a sentence: ")
for i in range(len(sentence)) :
   if sentence[i].isupper() :
      print(i) # the position, not the letter
```

Finding the **First** Match

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

```python
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 towards 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

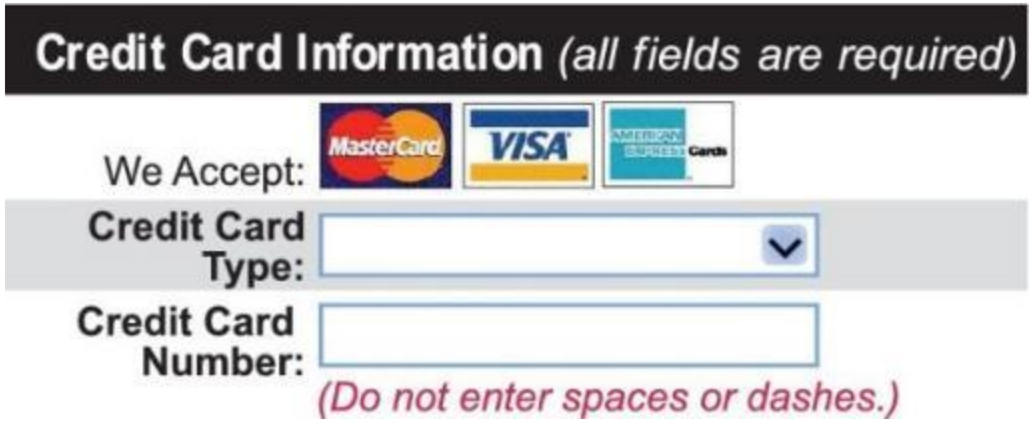
```
found = False
position = len(string) - 1
while not found and position >= 0 :
   if string[position].isdigit() :
      found = True
   else :
      position = position - 1
```

Building a New String (code)

- **The contents of a string cannot be changed.**
  - Python strings are “immutable”
  - But nothing prevents us from building a new string.

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?



Credit Card Information *(all fields are required)*

We Accept:

Credit Card Type:

Credit Card Number:

(Do not enter spaces or dashes.)

Building a New String (code)

- Here is a loop that builds a new string containing a credit card number with spaces and dashes removed:
  - We read the credit card number
  - We initialize a new string to the empty string (`= ""`)
  - We test each character in the user input
    - If the character is not a space or dash we append it to the new string (`+ char`)

```
userInput = input("Enter a credit card number: ")
creditCardNumber = ""
for char in userInput :
   if char != " " and char != "-" :
      creditCardNumber = creditCardNumber + char
```

Application: Random Numbers and Simulations

📘 4.9

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

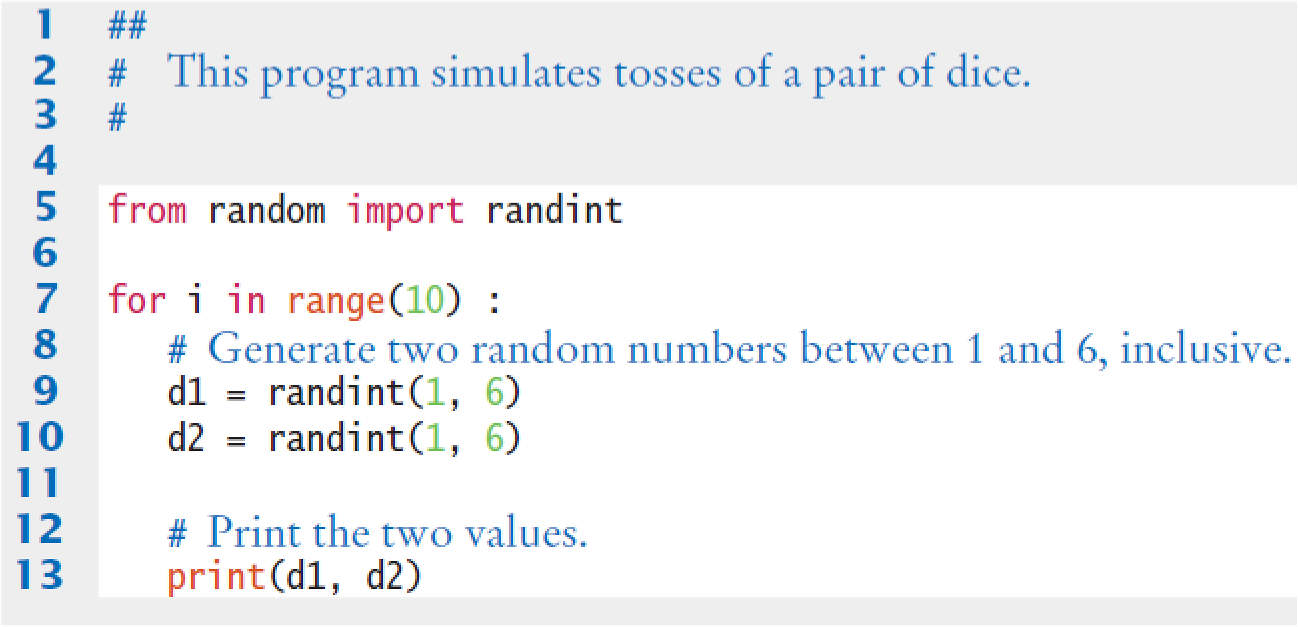
Generating Random Numbers

- 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
- Random generators are in the 'random' module
  - `from random import random`
  - https://docs.python.org/3/library/random.html
  - https://realpython.com/python-random/#prngs-in-python

Simulating Die Tosses

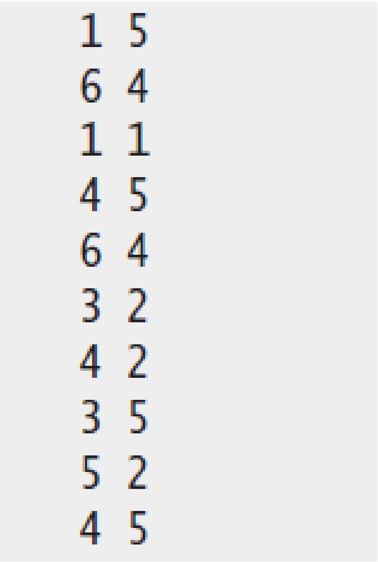
- Goal:
  - To generate a random integer in a given range we use the `randint()` function
  - `randint` has two parameters, the range (inclusive) of numbers generated

ch04/dice.py



```python
##
#  This program simulates tosses of a pair of dice.
#

from random import randint

for i in range(10) :
   # Generate two random numbers between 1 and 6, inclusive.
   d1 = randint(1, 6)
   d2 = randint(1, 6)

   # Print the two values.
   print(d1, d2)
```

Program Run

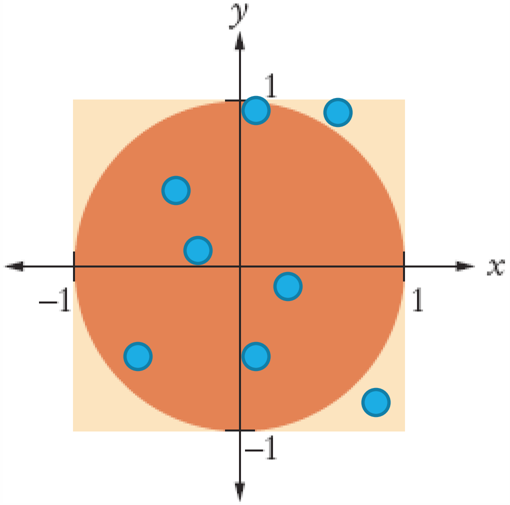


```
1 5
6 4
1 1
4 5
6 4
3 2
4 2
3 5
5 2
4 5
```

The Monte Carlo Method

- Used to find approximate solutions to problems that cannot be precisely solved
- Example: Approximate π using the relative areas of a circle inside a square
  - Uses simple arithmetic
  - Generate random points in [-1,1]x[-1,1]
  - `Hits` are inside circle, i.e. $x^2+y^2 \leqslant 1$
  - `Tries` are total number of tries
  - π is approximated as 4 x Hits / Tries
    - N.B. 4 is the area of the square, $\pi r^2 = \pi$ the area of the circle, so

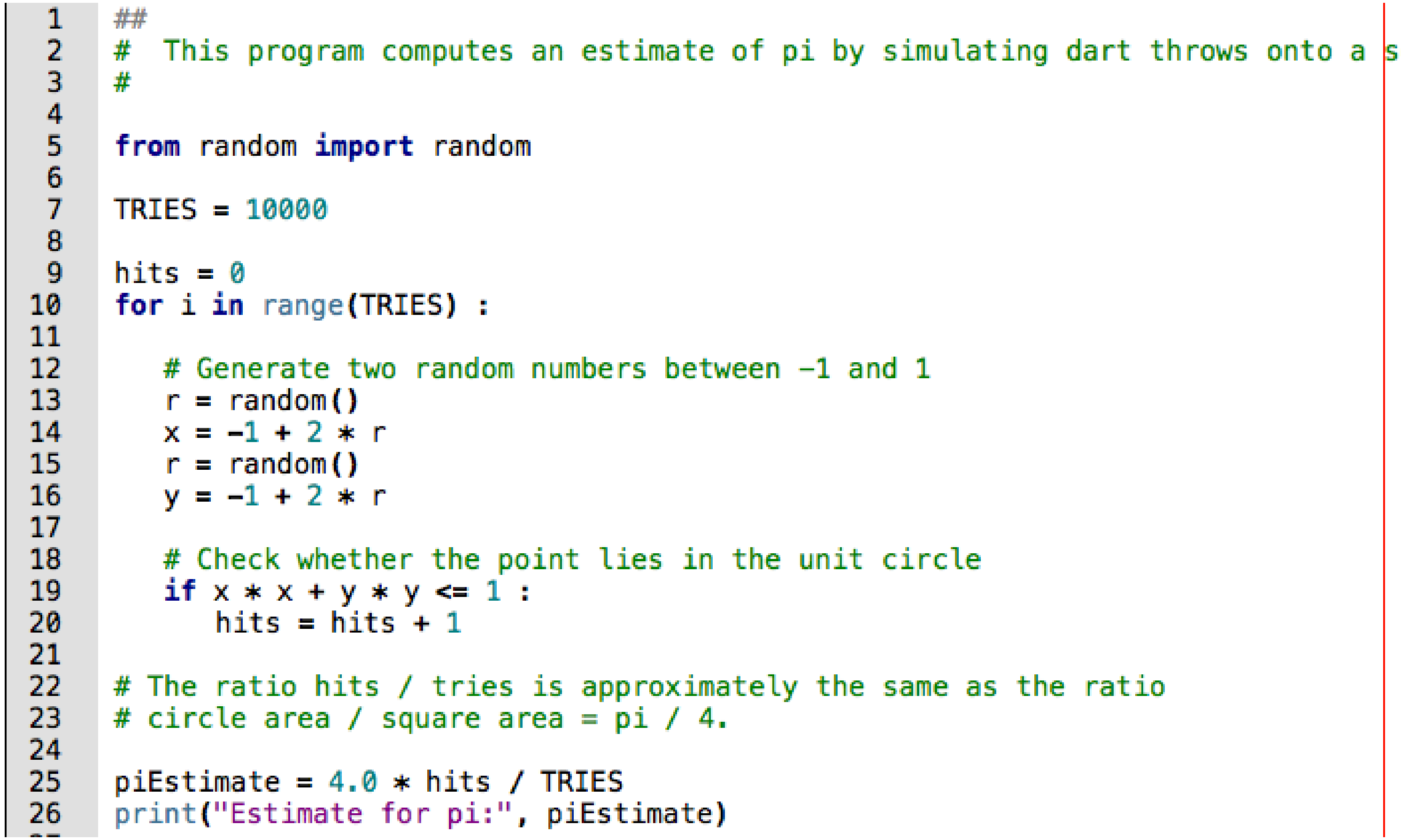
π:4 = `Hits` : `Tries`



The Monte Carlo Method

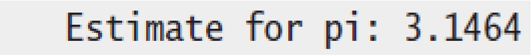
- Solution structure:
  - Generate `Tries` pairs (x,y) of random numbers (`random()`)
    - *(for i=1,..,N generate a pair)*
  - Rescale them to the [-1,1] interval
  - Verify if they are inside the circle
    - If yes -> increment `Hits`
  - At the end of the loop, `Hits/Tries` approximates pi/4

Monte Carlo Example



```
##
#  This program computes an estimate of pi by simulating dart throws onto a square
#

from random import random

TRIES = 10000

hits = 0
for i in range(TRIES) :

   # Generate two random numbers between -1 and 1
   r = random()
   x = -1 + 2 * r
   r = random()
   y = -1 + 2 * r

   # Check whether the point lies in the unit circle
   if x * x + y * y <= 1 :
      hits = hits + 1

# The ratio hits / tries is approximately the same as the ratio
# circle area / square area = pi / 4.

piEstimate = 4.0 * hits / TRIES
print("Estimate for pi:", piEstimate)
```

Program Run



```
Estimate for pi: 3.1464
```

Application: BlackJack

Blackjack

- Design and algorithm and write a program that plays Blackjack against the user. Then, write the corresponding Python code
- We assume there only one croupier and one player.
- The goal of BlackJack is to get as close as possible to 21, without going over it
  - It’s necessary to keep track of the sum of the values of the cards that the two participants (croupier and player) have been dealt

Rules of the game (simplified)

- Phase 1: the computer (croupier) takes a card (for itself) and shows it to the user
- Phase 2: the computer gives 2 cards to the user
- Phase 3: the player may choose to get another card, or to stop
  - This phase repeats until the player stops, or the sum of the cards is >21
  - In the second case, the player loses
- Phase 4: if the user didn’t lose, the computer plays, with a simple strategy
  - If the sum of its cards is < 17, then **take another card**
  - If it’s >= 17, **stop**
  - If it’s > 21, the computer **loses**

End game

- If the croupier goes over 21, the player wins
- If both stop, then the two sums must be compared. The player with a higher sum wins the round. The croupier wins in case of parity.

Simplifying Assumptions

- We deal from an **infinite pool of cards**
  - At every turn, any card had the same probability
  - **NOTE: This is not very realistic, but simplifies the program**

- Cards have values from 1 to 10
  - Just deal with integers
  - Aces count 1 (not 11)
  - Don’t care about figures
  - Don’t care about suits ♠♥♦♣

https://en.wikipedia.org/wiki/Playing_cards_in_Unicode

Extra (Improvements)

- Let the players play more than one round (and count points!)
- The Ace 🂡🂱🃁🃑 may have the value 1 or 11, according to the user preference
- When a card has been dealt, it’s no longer in the deck, so the probability of dealing another card with the same value decreases

Summary

Summary: Two Types of Loops

- `while` loops
- `for` loops
- `while` loops are very versatile
- Uses of the `for` loop:
  - The `for` loop can be used to iterate over the contents of any container.
  - A `for` loop can also be used as a count-controlled loop that iterates over a range of integer values.

Summary

- Each loop requires the following steps:
  - Initialization (setup variables to start looping)
  - Condition (test if we should execute loop body)
  - Update (change something each time through)
  - A `for-range` does it automatically, in a `while` it's the programmer's responsibility

- A loop executes instructions repeatedly while a condition is True.

- An off-by-one error is a common error when programming loops.
  - Think through simple test cases to avoid this type of error.

Summary

- A sentinel value denotes the end of a data set, but it is not part of the data.
- You can use a boolean variable to control a loop.
  - Set the variable to `True` before entering the loop
  - Set it to `False` to leave the loop.
- Loops can be used in conjunction with many string processing tasks
- In a simulation, you use the computer to simulate an activity.
  - You can introduce randomness by calling the random number generator.