ISYS90088
Introduction to Application Development

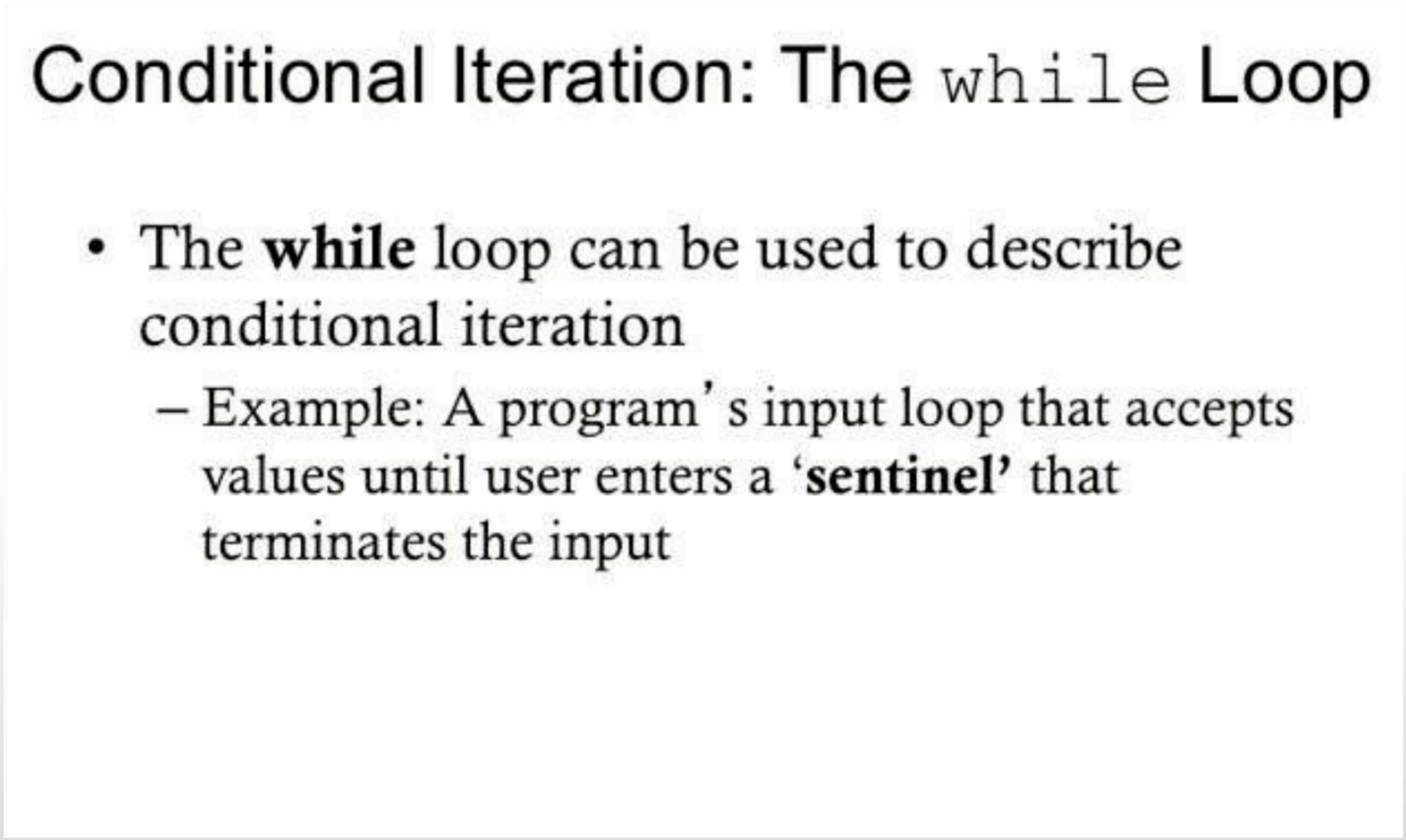
Week 6 lectures – Continued from week 5:
While and for statements; formatting

Department of Computing and Information Systems

University of Melbourne

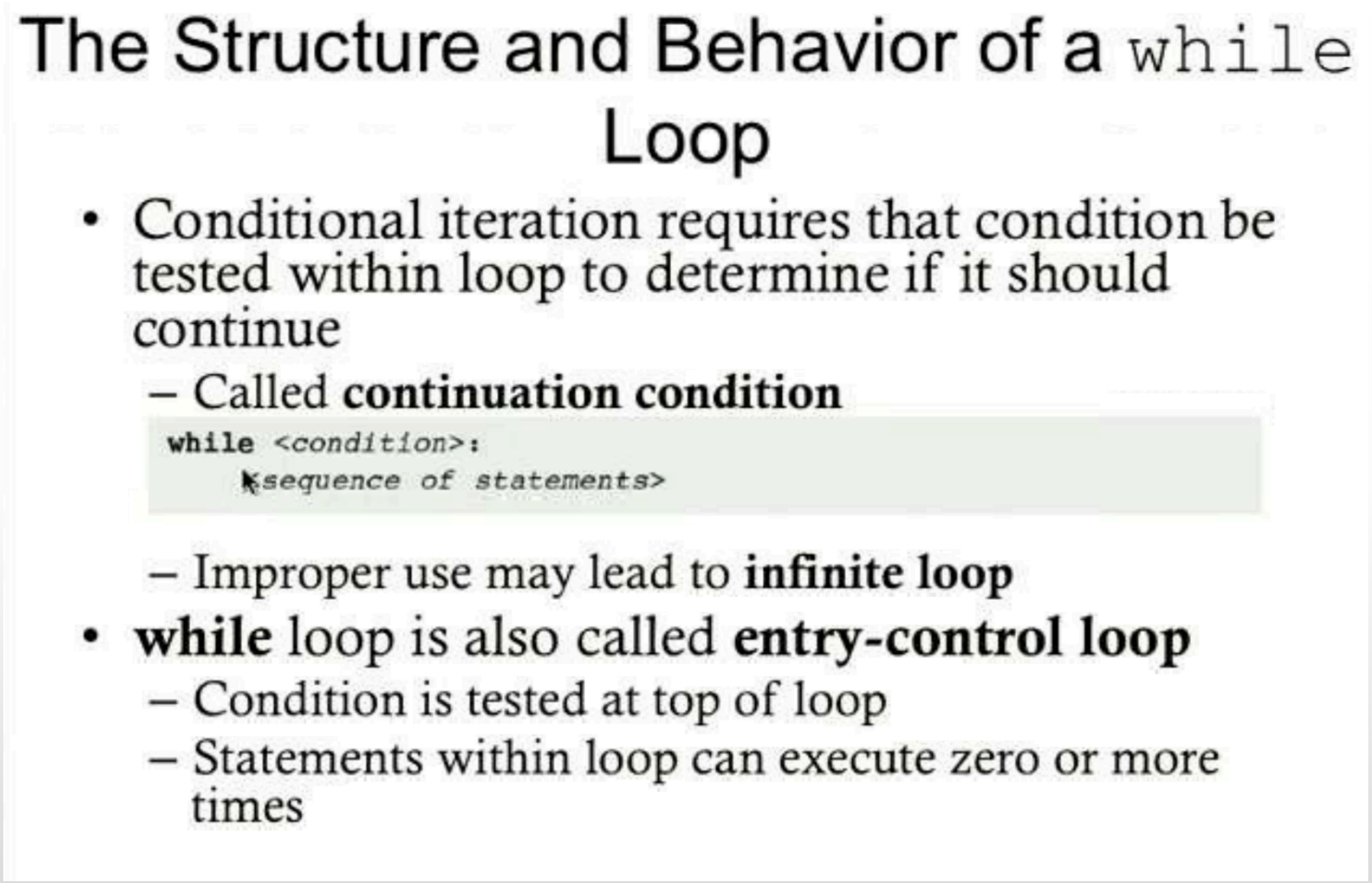
Semester 2 , 2017

Dr. Thomas Christy



Conditional Iteration: The `while` Loop

- The **while** loop can be used to describe conditional iteration
  - Example: A program's input loop that accepts values until user enters a '**sentinel'** that terminates the input



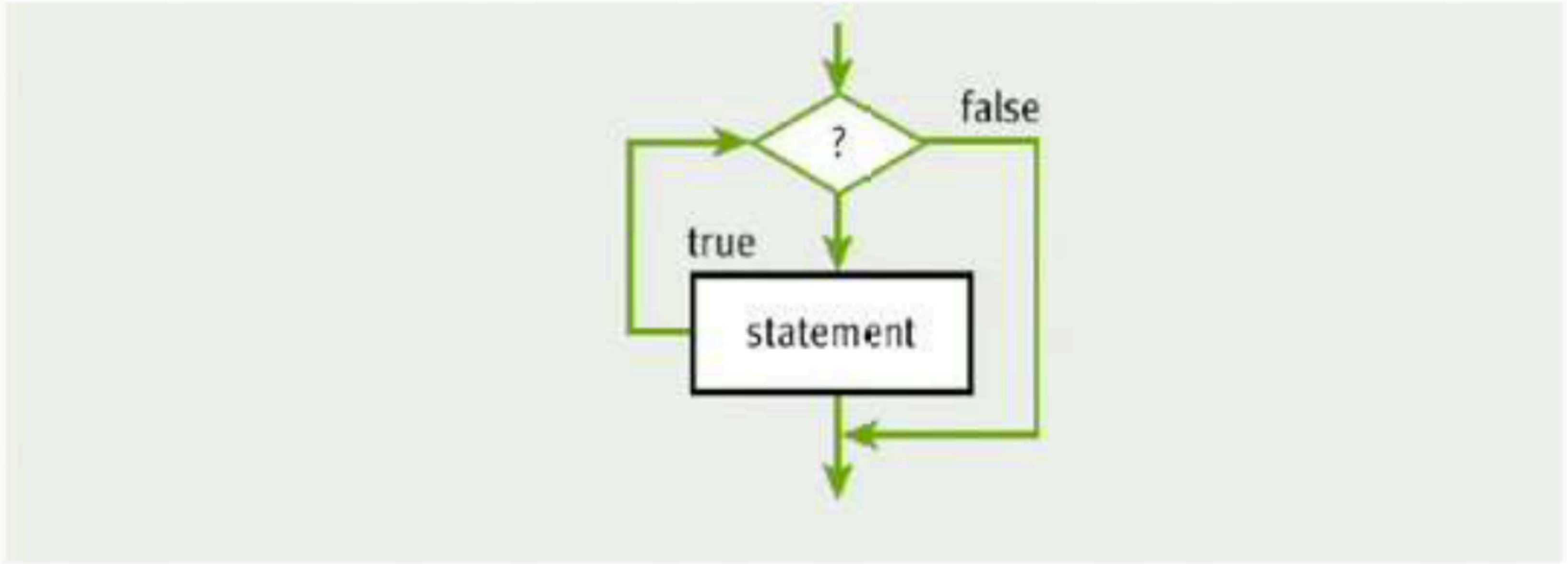
The Structure and Behavior of a `while` Loop

- Conditional iteration requires that condition be tested within loop to determine if it should continue
  - Called **continuation condition**

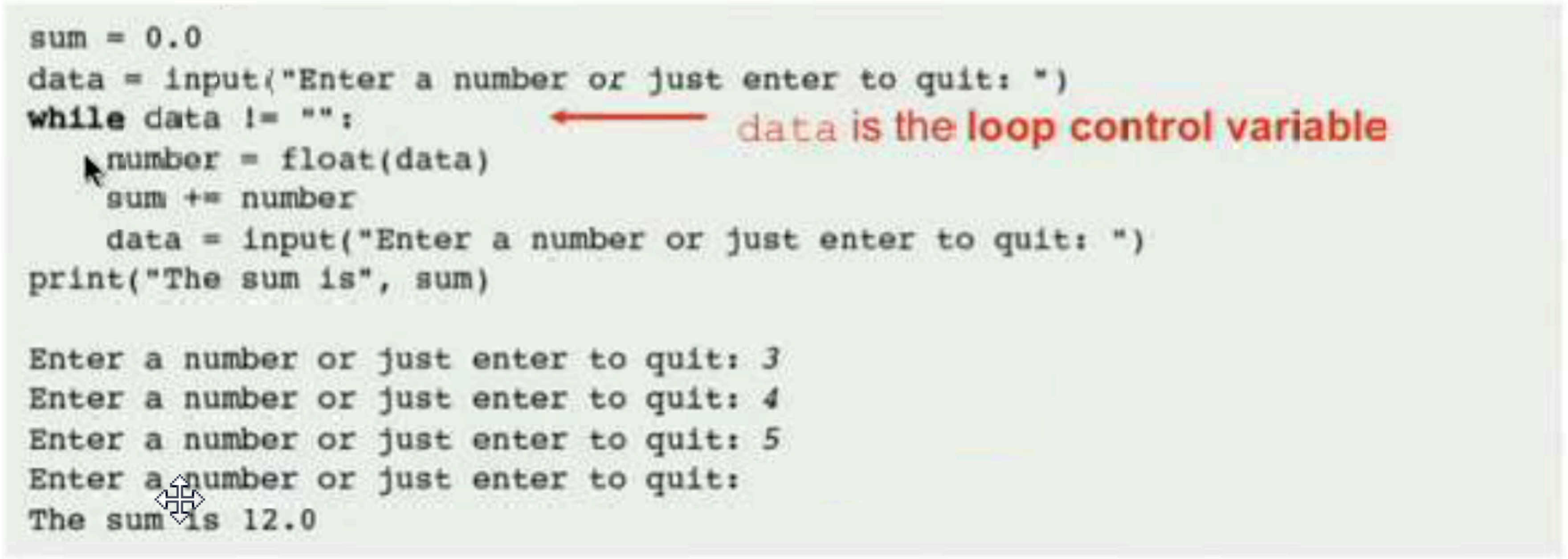
```
while <condition>:
    <sequence of statements>
```

  - Improper use may lead to **infinite loop**
- **while** loop is also called **entry-control loop**
  - Condition is tested at top of loop
  - Statements within loop can execute zero or more times

The Structure and Behavior of a `while` Loop (continued)

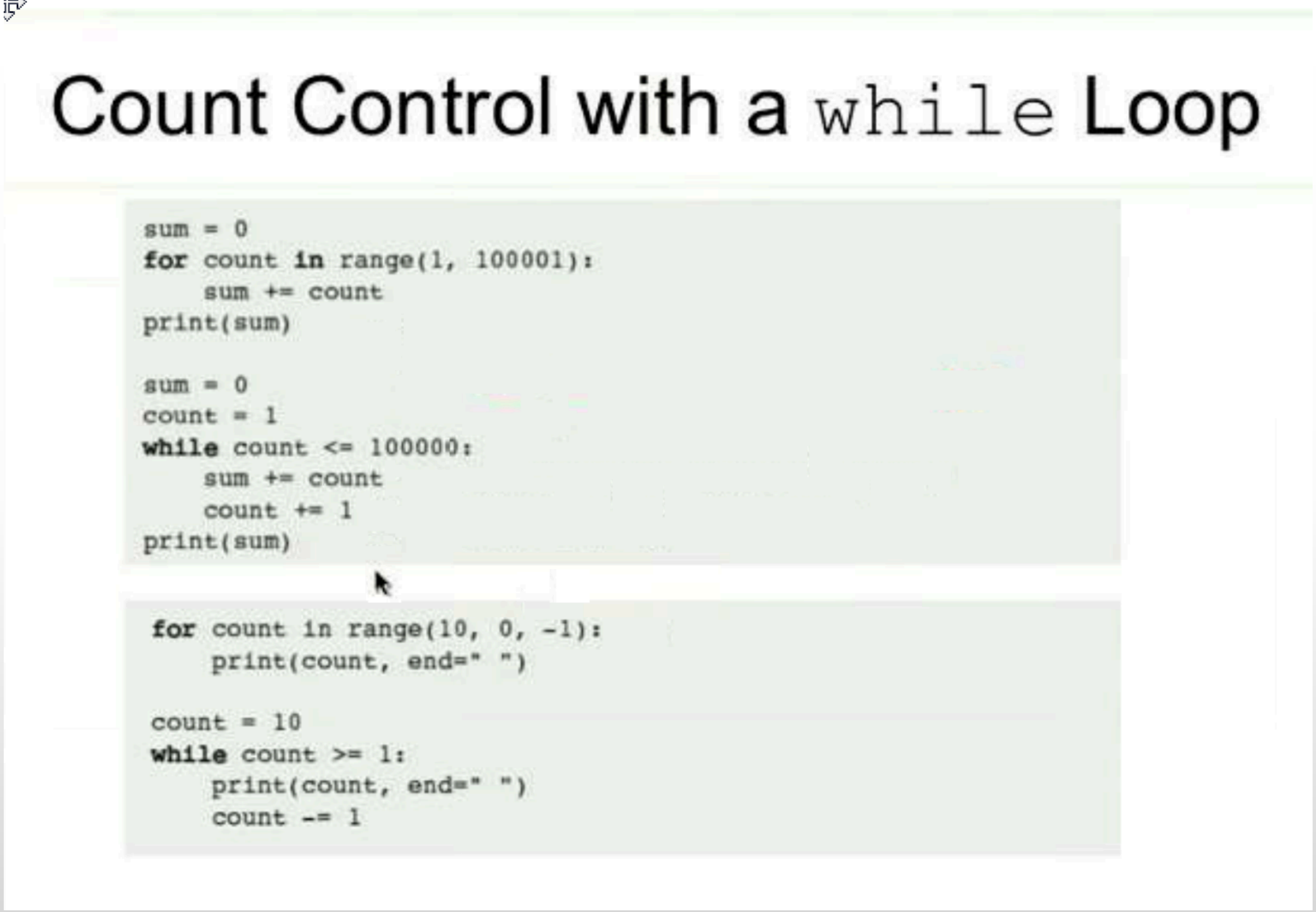


The Structure and Behavior of a `while` Loop (continued)



```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", sum)

Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit:
The sum is 12.0
```

`data` **is the loop control variable**



Count Control with a `while` Loop

```
sum = 0
for count in range(1, 100001):
    sum += count
print(sum)

sum = 0
count = 1
while count <= 100000:
    sum += count
    count += 1
print(sum)
```

```
for count in range(10, 0, -1):
    print(count, end=" ")

count = 10
while count >= 1:
    print(count, end=" ")
    count -= 1
```

Loop control statements

Loop Control Statements

- Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.
- Python supports the following control statements:

break statement: Terminates the loop statement and transfers execution to the statement immediately following the loop.

continue statement: Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

pass statement

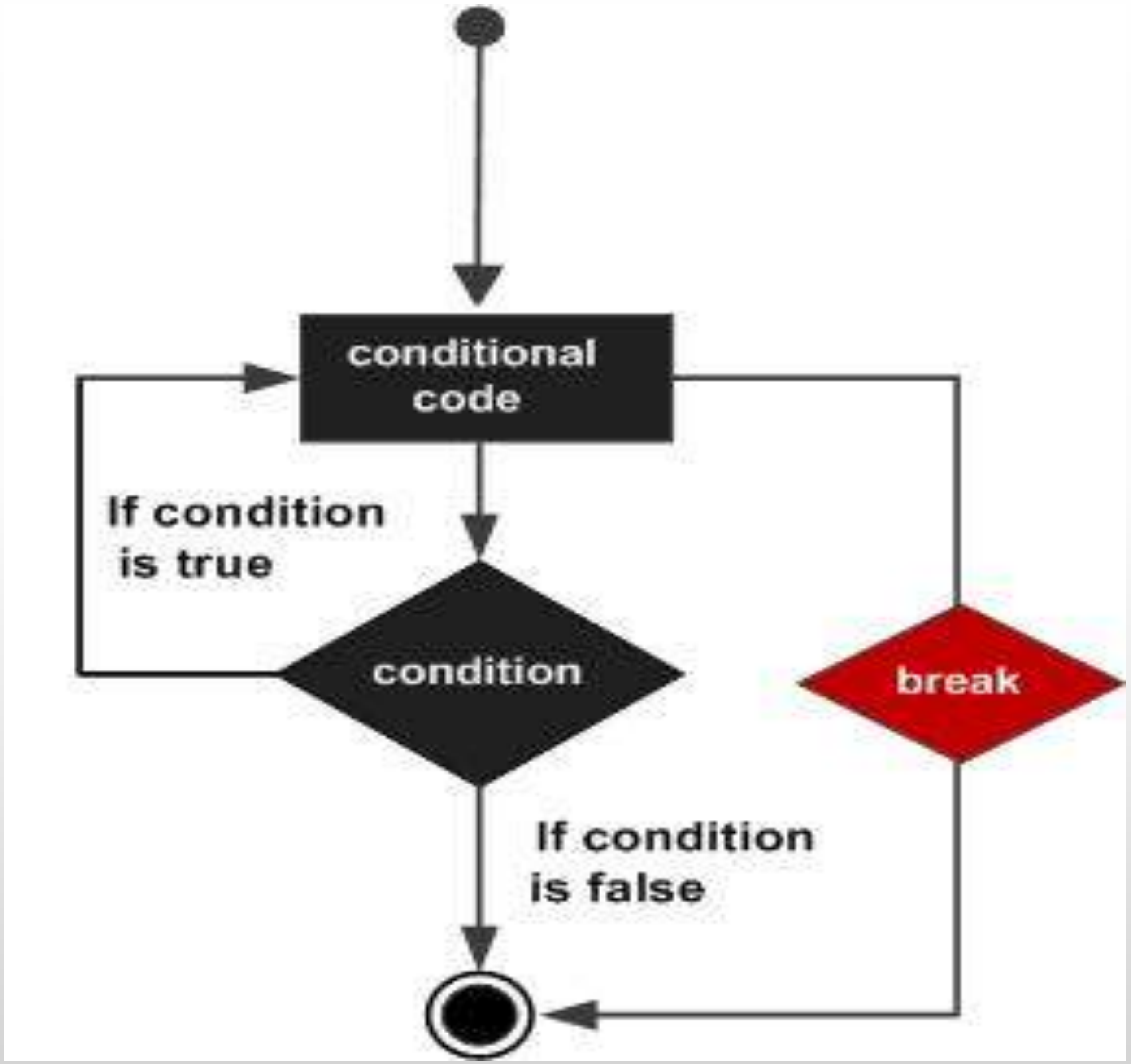
break statement

- It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.
- If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

```
>>> break
```

break statement



The `while True` Loop and the `break` Statement

```
done = False
while not done:          ← a while True loop with a delayed exit
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:      ← loop's termination condition
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number)   # Just echo the valid input
```

- Alternative: Use a boolean and also a break stmt

```
while True:          ← a while True loop with a delayed exit
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:      ← loop's termination condition
        break        ← causes an exit from the loop
    else:
        print("Error: grade must be between 100 and 0")
print(number)   # Just echo the valid input
```

break statement

- It terminates the current loop and resumes execution at the next statement.

- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop.

- The break statement can be used in both while and for loops.

- If you are using nested loops, the break statement stops the execution of the innermost loop and starts executing the next line of code after the block.

Syntax:

```
>>> break
```

continue statement

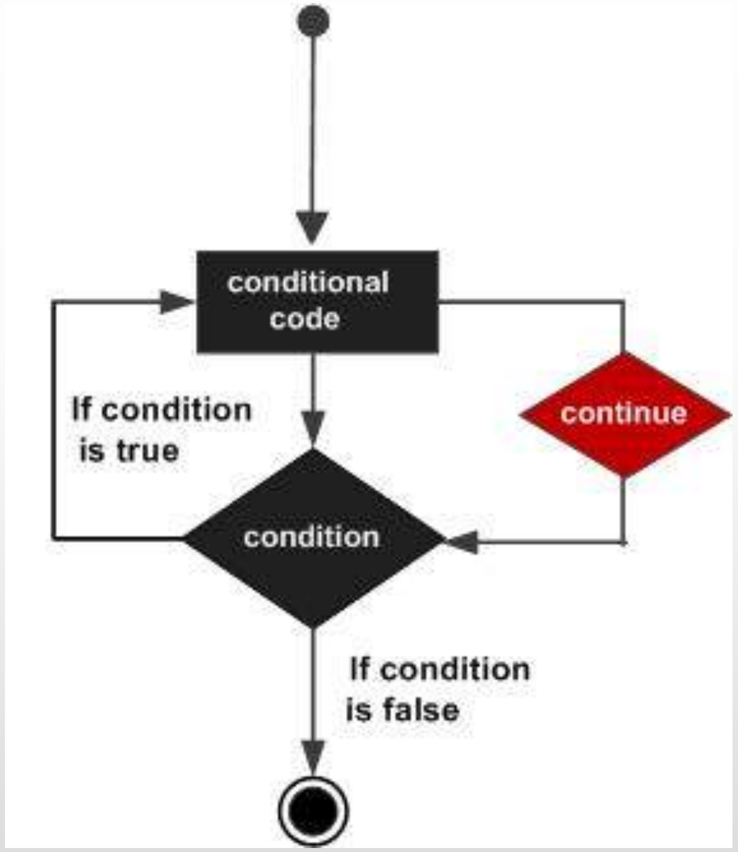
- It returns the control to the beginning of the while loop.. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

- The continue statement can be used in both while and for loops.

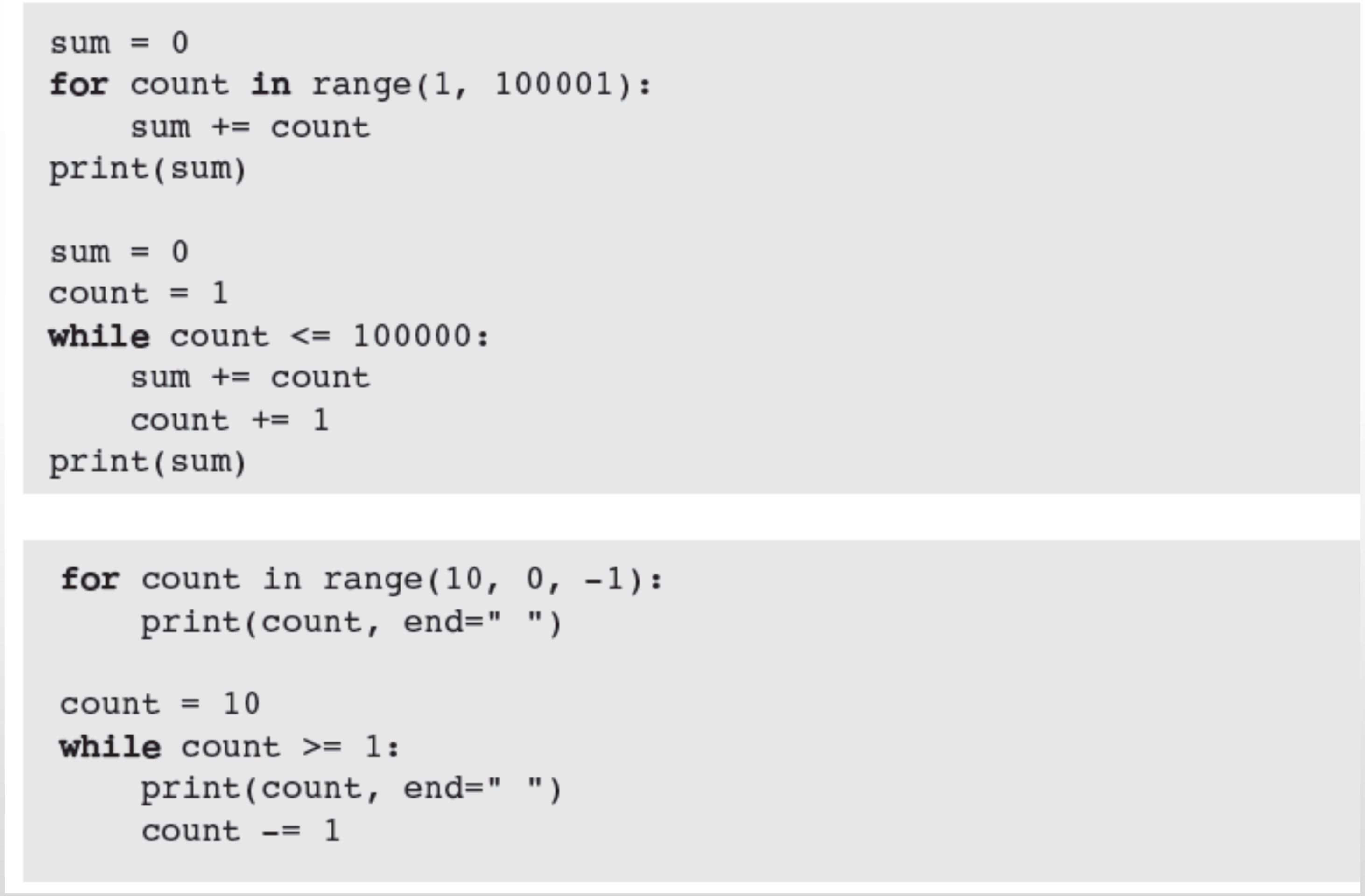
Syntax:

```
>>> continue
```

continue statement



Reflect: `for` **`loop to`** `while` **`loop`**



```
sum = 0
for count in range(1, 100001):
    sum += count
print(sum)

sum = 0
count = 1
while count <= 100000:
    sum += count
    count += 1
print(sum)
```

```
for count in range(10, 0, -1):
    print(count, end=" ")

count = 10
while count >= 1:
    print(count, end=" ")
    count -= 1
```

Example: try writing a for loop for this while

Example: This code checks whether a word contains digits or not.

```
word = input('enter a word:')
found_digit = False
i = 0
while (not found_digit) and i < len(word):
    if word[i].isdigit():
        found_digit = True
        print("The word contains digits!")
    i = i + 1
if not found_digit:
    print("The word does not contain digits!")
```

Example: try writing a for loop for this while

Example: This code checks whether a word contains digits or not.

```
word = input('enter a word:')
for i in range (0, len(word)):
    if word[i].isdigit():
        print("The word contains digits!")
        break
if not (word[i].isdigit()):
    print("The word does not contain digits!")
```

When to use : `for` and `while` loop

Simplest way to differentiate between the **for** and the **while**:

- we usually use **for** when there is a known number of iterations, and use **while** constructs when the number of iterations in not known in advance.
- **while** loops are slightly "fiddlier" than **for** loops, in that we need to set up a test in the **while** condition, and make sure to update the variable in the test appropriately in the body of our code.
- In programming, "fiddlier"/more lines of code tends to correlate with "greater margin for error", and as such **for** loops should be your default choice.
- expect/aim to use **for** much more than **while**.

Nested `for` loops

Syntax for nested `for`**:**

```
for iterating_var in sequence:
    for iterating_var in sequence:
        statements(s)
statements(s)
```

Syntax for nested `while`:

```
while <condition or expression>:
    while <condition or expression>:
        statement(s)
statement(s)
```

Nested `for` loops

#simple example to illustrate the nested for

```
n = int(input(‘enter a number:’))
for i in range(1,n):
     for j in range(1,n):
         print (i, j)
print("good bye")
```

Nested loops – when do we use it?

Example: For every word (from a list), look at every character in that word. This construct might look like this:

```
listofWord = ['cat', 'dog', 'fish']
for word in listofWord:
      for letter in word:
            < do something....>
```

Examples: A simple nested for loop

```
"""
Example of code that draws out the following: say n = 5, then your drawing will look like:
#
##
###
####
"""
symbol = '#'
number = int(input ('enter a number:'))
for x in range(1, number):
    s = ""
    for y in range(x):
        s += symbol
    print (s)
```

Examples: A simple nested `for` loop

Example code that checks whether numbers between 1 and 20 are prime.

## to calculate if a number is prime or not

```
for num in range(1,20):  #to iterate between 1 to 20
   for i in range(2,num): #to iterate on the factors of the number
      if num%i == 0:      #to determine the first factor
         print (num, 'is not prime')
         break
   else:
      print (num, 'is a prime number')
```

- Try doing this using a while as home work!!!!!

Work this one out!

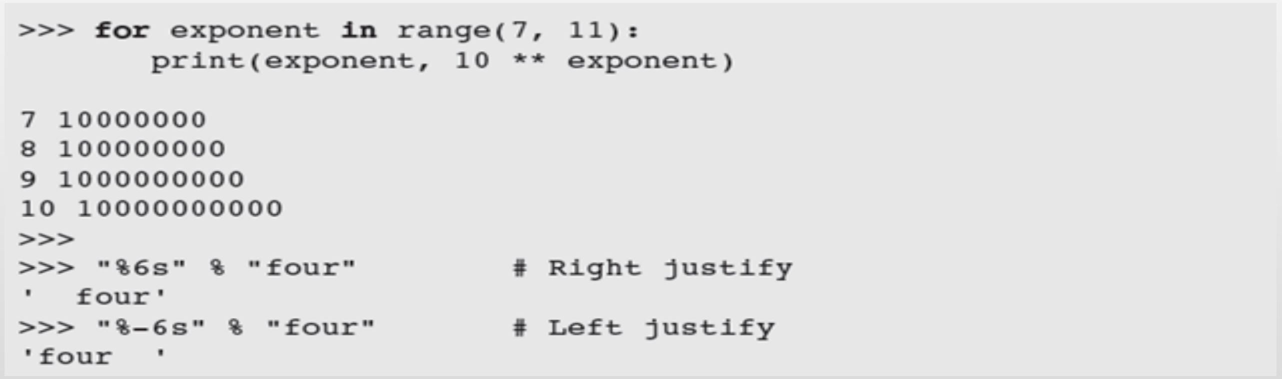
```
""

The following example illustrates the combination of an else
statement with a for statement that searches for prime numbers
from 1 through 20.And prints whether the number is prime. It
also calculates and prints the factors if the number is not
prime

"””

for num in range(XXX,XXX):

   for i in range(2,XXX): #to iterate on factors of number

      if XXX == 0:      #to determine the first factor

         j = XXX       #to calculate the second factor

         print (’XXX equals XXX * XXX' % (num,i,j))

         break

   else:

         print (num, 'is a prime number')
```

Work this one out!

```
""

The following example illustrates the combination of an else
statement with a for statement that searches for prime numbers
from 1 through 20.And prints whether the number is prime. It also
calculates and prints the factors if the number is not prime

"""

for num in range(1,20):  #to iterate between 1 to 20
   for i in range(2,num): #to iterate on factors of number
      if num%i == 0:      #to determine the first factor
         j = num/i        #to calculate the second factor
         print ('%d equals %d * %d' % (num,i,j))
         break
   else:
         print (num, 'is a prime number')
```

Formatting Text for Output

- Many data-processing applications require output that has **tabular format**
- **Field width**: Total number of data characters and additional spaces for a datum in a formatted string



```
>>> for exponent in range(7, 11):
        print(exponent, 10 ** exponent)

7 10000000
8 100000000
9 1000000000
10 10000000000
>>>
>>> "%6s" % "four"          # Right justify
'  four'
>>> "%-6s" % "four"         # Left justify
'four  '
```

Formatting Text for Output (continued)

```
<format string> % <datum>
```

- This version contains **format string**, **format operator %**, and single data value to be formatted
- To format integers, letter **d** is used instead of **s**

- To format sequence of data values:

```
<format string> % (<datum-1>, …, <datum-n>)
```

```
>>> for exponent in range(7, 11):
        print("%-3d%12d" % (exponent, 10 ** exponent))

7      10000000
8     100000000
9    1000000000
10  10000000000
```

Formatting Text for Output (continued)

- To format data value of type **float**:

```
%<field width>.<precision>f
```

where .***<precision>*** is optional

- Examples:

```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
>>>
```

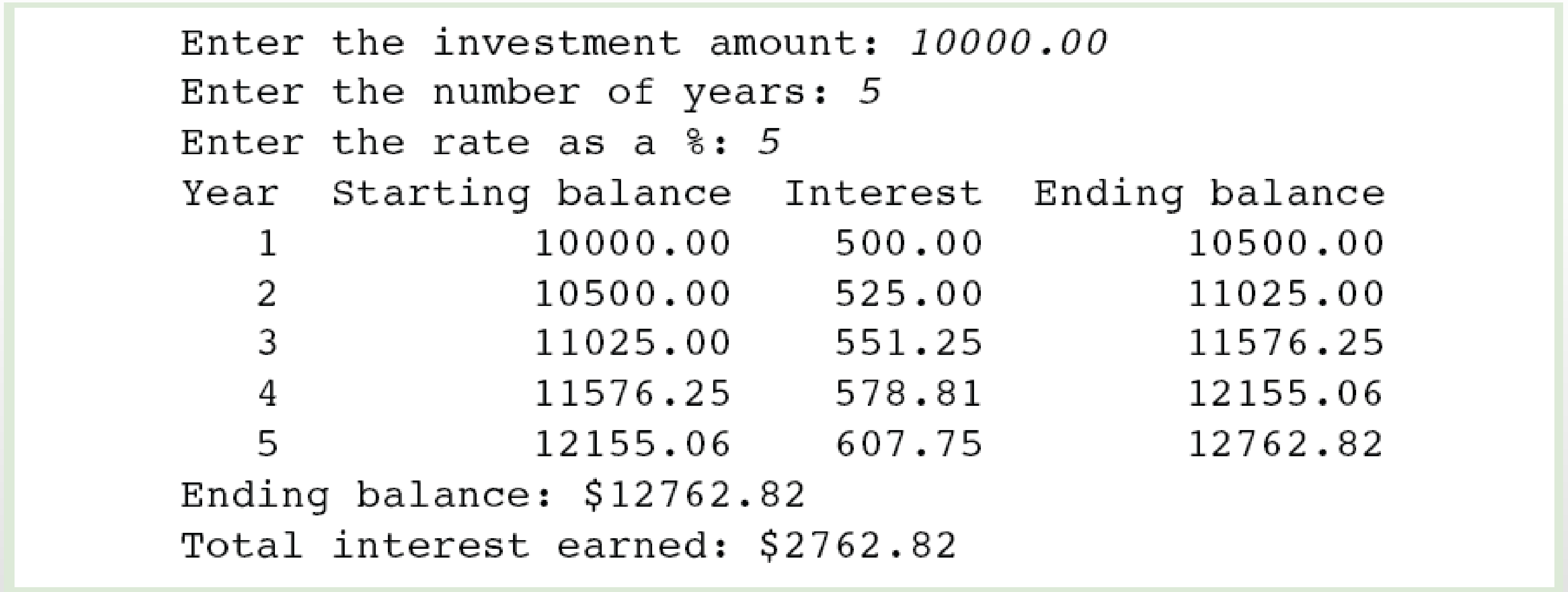
```
>>> "%6.3f" % 3.14
' 3.140'
```

Case Study: An Investment Report

- Request:
  - Write a program that computes an investment report

Case Study: An Investment Report (continued)

- Analysis:



```
Enter the investment amount: 10000.00
Enter the number of years: 5
Enter the rate as a %: 5
Year  Starting balance  Interest  Ending balance
   1          10000.00    500.00        10500.00
   2          10500.00    525.00        11025.00
   3          11025.00    551.25        11576.25
   4          11576.25    578.81        12155.06
   5          12155.06    607.75        12762.82
Ending balance: $12762.82
Total interest earned: $2762.82
```

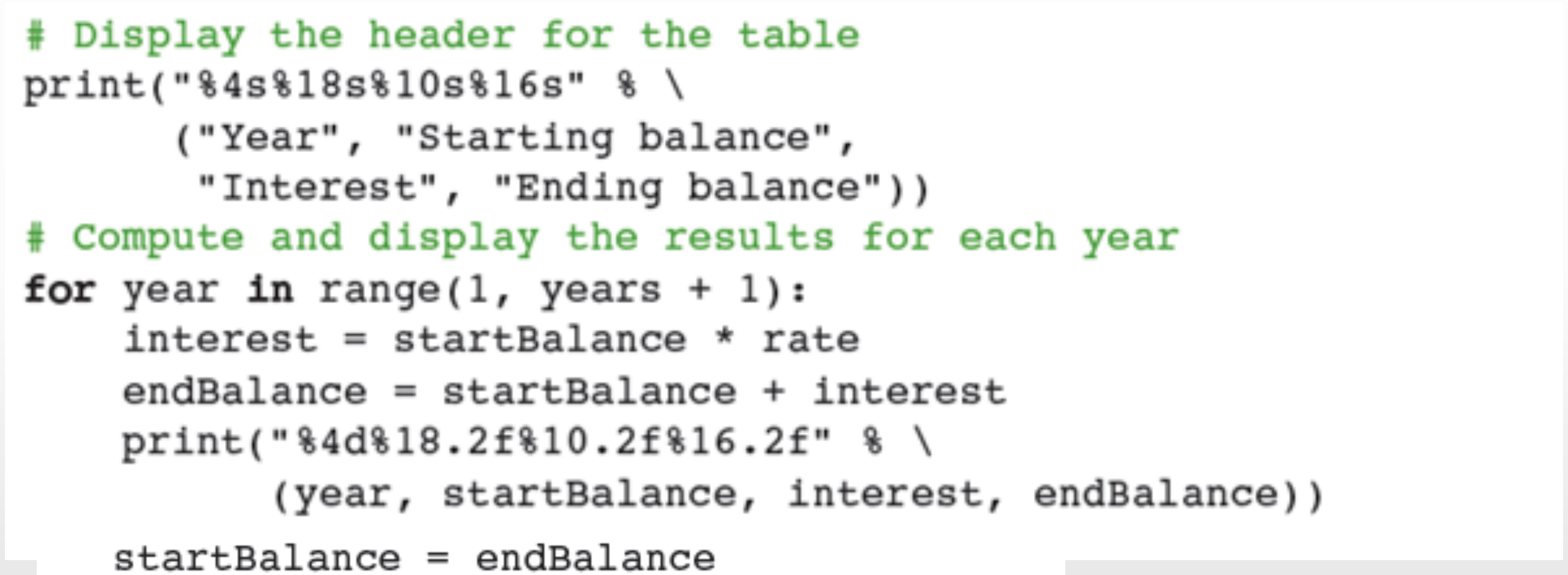
[FIGURE 3.1] The user interface for the investment report program

Case Study: An Investment Report (continued)

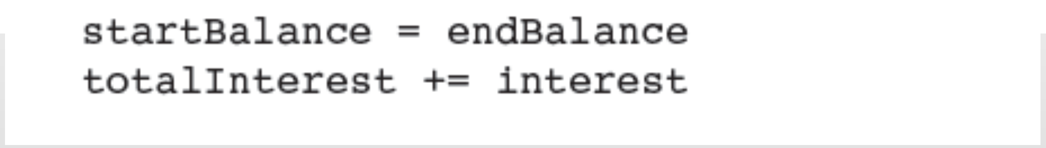
- Design:
  - Receive the user's inputs and initialize data
  - Display the table's header
  - Compute results for each year and display them
  - Display the totals

Case Study: An Investment Report (continued)

- Coding:



```python
# Display the header for the table
print("%4s%18s%10s%16s" % \
      ("Year", "Starting balance",
       "Interest", "Ending balance"))
# Compute and display the results for each year
for year in range(1, years + 1):
    interest = startBalance * rate
    endBalance = startBalance + interest
    print("%4d%18.2f%10.2f%16.2f" % \
          (year, startBalance, interest, endBalance))
```



```python
    startBalance = endBalance
    totalInterest += interest
```

Formatting Text for Output (continued)

- When the field width is positive, the datum is right justified
- When the field width is negative, the datum is left justified
- If the field width is less that or equal to the datum’s print length in characters, no justification is added.

Formatting Text for Output (continued)

- Examples:

```
%<field width>.<precision>f
```

```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
>>>
```

```
>>> "%6.3f" % 3.14
' 3.140'
```

Note: the width includes the place for the decimal point

Formatting: examples

```
>>>amount  = 24.325
>>>print('your salary is $%0.2f' % amount)

>>>print('The area is %0.1f' % amount)

>>>print('%10.4f' % amount)

>>>print('%.5s' % ('tropical'))

>>>print(‘%5s’ % (‘tropical’))

>>>print('%5s' % ('trop'))
```

Formatting: examples

```
>>>amount = 24.325
>>>print('your salary is $%0.2f' % amount)
24.32
>>>print('The area is %0.1f' % amount)
24.3
>>>print('%10.4f' % amount)
   24.3250
>>>print('%.5s' % ('tropical'))
tropi
>>>print(‘%5s’ % (‘tropical’))
tropical
>>>print('%5s' % ('trop'))
 trop
```

Example : formatting

- Write a code segment that displays the values of the integers x, y, z on a single line, such that each value is right-justified in six columns.
- Then try the same as above but left justified
- Then try out the same as above but with the values of x, y and z printed on separate lines

Example : formatting

- Write a code segment that displays the values of the integers x, y, z on a single line, such that each value is right-justified in six columns.

```
>>>print("%6d%6d%6d" % (x, y, z))
```

- Then try the same as above but left justified

```
>>>print("%-6d%-6d%-6d" % (x, y,
z))
```

- Then try out the same as above but with the values of x, y and z printed on separate lines

```
>>>print("%6d\n%6d\n%6d" % (x, y, z))
>>>print("%-6d\n%-6d\n%-6d" % (x, y, z))
```

Formatting multiple values

Syntax:

```
print (<string> % (num, num ...))
```

Note: same number of formatting specifiers as values are needed for formatting

```
>>>val1 = 6.7891234
>>>val2 = 1.2345678
>>>val3 = 123456789.123456789
>>>print('values are %.1f and %.3f and %6.2f' %(val1, val2, val3))
```

Formatting values: exercise – try this one!

```
>>>my_value = 7.2386
>>>print('%0.2f' % my_value)

>>>amt = 5000.0
>>>m_pay = amt/12.0
>>>print('%0.2f' % m_pay)

>>>my_new_value = 1.123456789

>>>print('%.2f' % my_new_value)
>>>print('%.4f' % my_new_value)
>>>print('%6.2f' % my_new_value)
```

Formatting strings

```
>>> s = ‘mysterious’
# 7 characters in the string
>>>print('%.*s' % (7, s))

# two characters in the string
>>>print('%.*s' % (2, s))

###exponent
>>>print('%10.3e' % (2000.345))

>>>print('%10.2E' % (3456.234))

>>> x = 2000000
>>> print('%10e' % x)
```

Formatting strings

```
>>> s = ‘mysterious’
# 7 characters in the string
>>>print('%.*s' % (7, s))
mysteri
# two characters in the string
>>>print('%.*s' % (2, s))
my
###exponent
>>>print('%10.3e' % (2000.345))
 2.000e+03
>>>print('%10.2E' % (3456.234))
  3.46E+03
>>> x = 2000000
>>> print('%10e' % x)
2.000000e+06
```

Formatting: examples (new styling (vs) old style of formatting

- Old style syntax:

```
<format string> % (<datum-1>, …, <datum-n>)
```

- New formatting style syntax in general:

```
<format string> % (<datum-1>, …, <datum-n>)
```

Old: *<format string> : **format**(<datum-1>, …, <datum-n>)*

New:

```
%<field width>.<precision>f
```

{':<field width>.<precision>f'}

Formatting: examples (new styling)

#Examples for formatting using the format()

# using <, >, ^ and a filler

```
>>>print('{:_<10}'.format('test'))
>>>print('{:_^10}'.format('test'))
>>>print('{:_>10}'.format('test'))
>>>print('{:*>10}'.format('test'))
```

Check other examples – file uploaded on LMS

Formatting: examples (new styling)

```
count = 10
total = 100
print(‘The number contains {}digits’. format(count))
print(‘The digits sum to {}’. format(total))
```

Examples and side notes – lots of them to check out! (see uploaded on the LMS.

Note: There are a few rule changes when you use the format() – new style of formatting

- Check the s format alignment! (example2_format())

Formatting: new (vs) old approach

Signed numbers - By default only negative numbers are prefixed with a sign.

#Old

```
>>>print('%+d' % (60))
>>>print('%d' % ((-40)))
```

#New

```
>>>print('{:+d}'.format(60))
>>>print('{:d}'.format((-40)))
```

Formatting: examples

# example that uses date and time method

```
>>>from datetime import datetime
>>>print('{:%Y-%m-%d %H:%M}'.format(datetime(2016, 2, 10, 4, 30)))
```

#example that uses a list

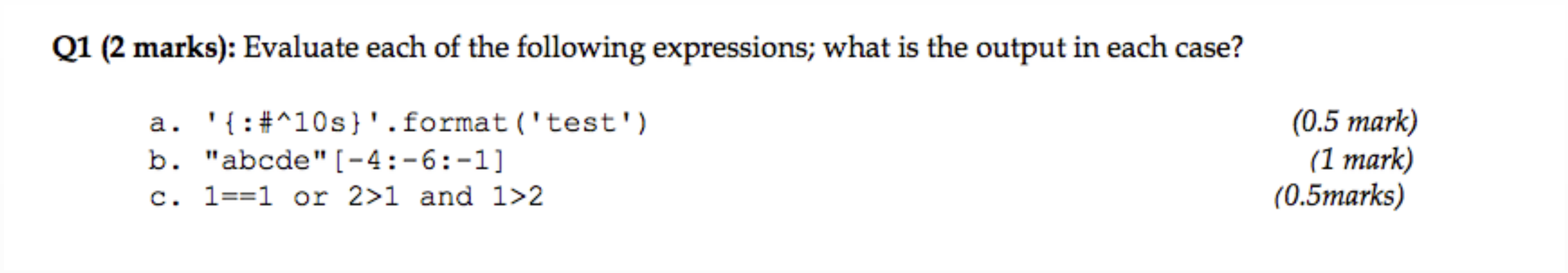
```
  data = [4, 8, 15, 16, 23, 42, 34, 65, 12]
  print('{d[4]} {d[5]}'.format(d=data))
```

Sample Mid-semester test: Semester 2, 2016

- **Authorized materials:** No materials are authorized. Calculators are not permitted.
- **Instructions to invigilators:** Students may not remove any part of the examination paper from the examination room. Students should be supplied with the exam paper and a script book, and with additional script books on request.
- **Instructions to students:** All questions should be answered by writing a brief response or explanation on the lined pages in the script book. The reverse side of any page may be used to make rough notes, or prepare draft answers. Please write your student ID below and on your script book. When you are finished, place the exam paper inside the front cover of the script book.

  - This test is worth 10 marks, which will count as 10% of your final grade.
  - Unreadable answers will be deemed wrong.
  - Start your answer to each question on a new page.
  - Be sure to clearly number your answers.
  - Use a blue or black pen (not a pencil).

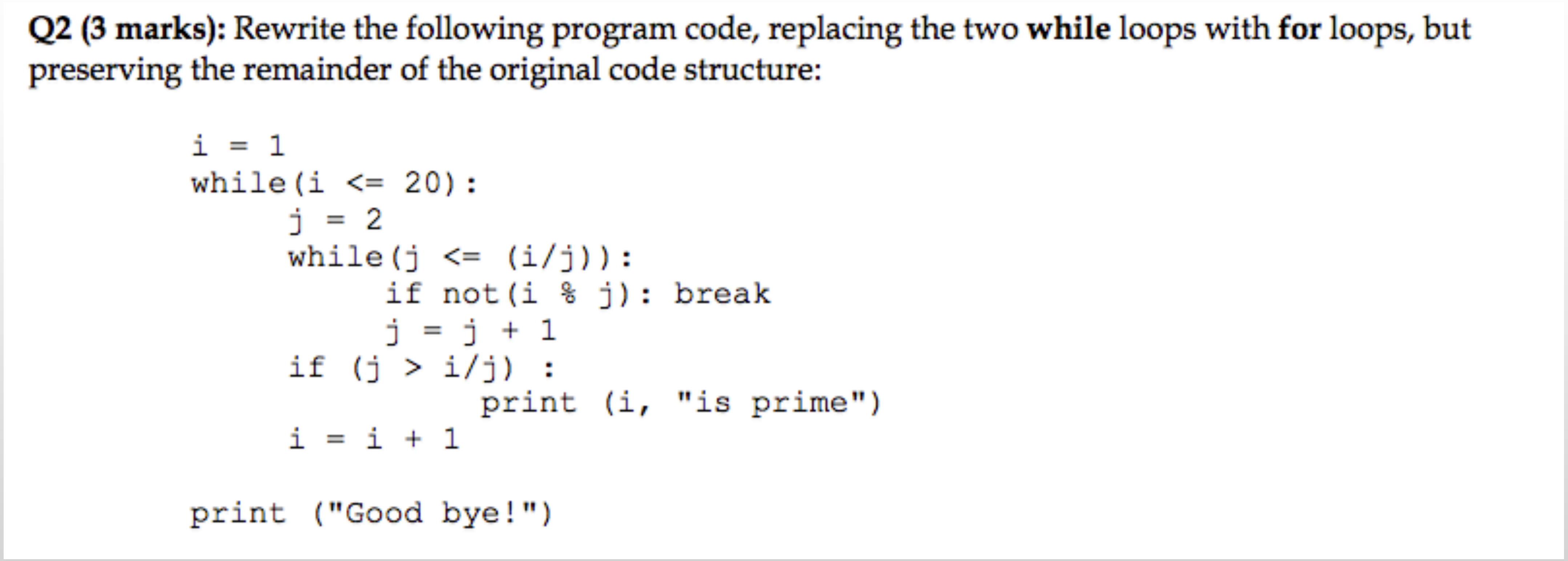
Sample Test



Q1 (2 marks): Evaluate each of the following expressions; what is the output in each case?

```
a. '{:#^10s}'.format('test')
b. "abcde"[-4:-6:-1]
c. 1==1 or 2>1 and 1>2
```

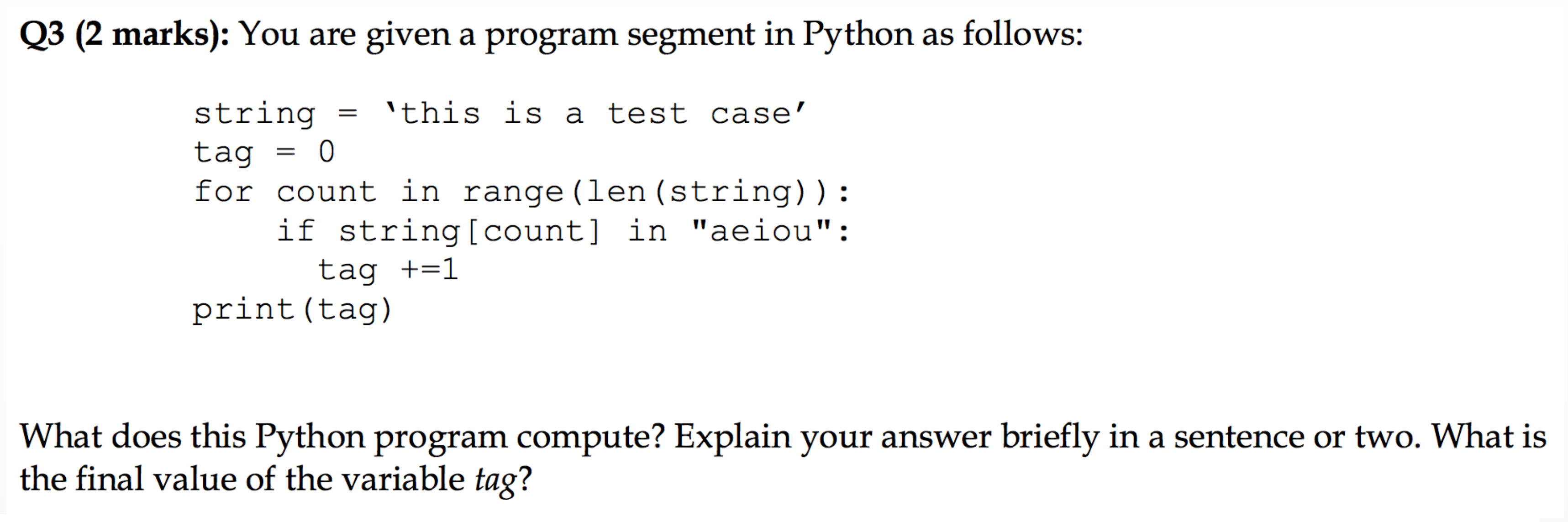
(0.5 mark)
(1 mark)
(0.5marks)



Q2 (3 marks): Rewrite the following program code, replacing the two **while** loops with **for** loops, but preserving the remainder of the original code structure:

```
i = 1
while(i <= 20):
     j = 2
     while(j <= (i/j)):
          if not(i % j): break
          j = j + 1
     if (j > i/j) :
               print (i, "is prime")
     i = i + 1

print ("Good bye!")
```

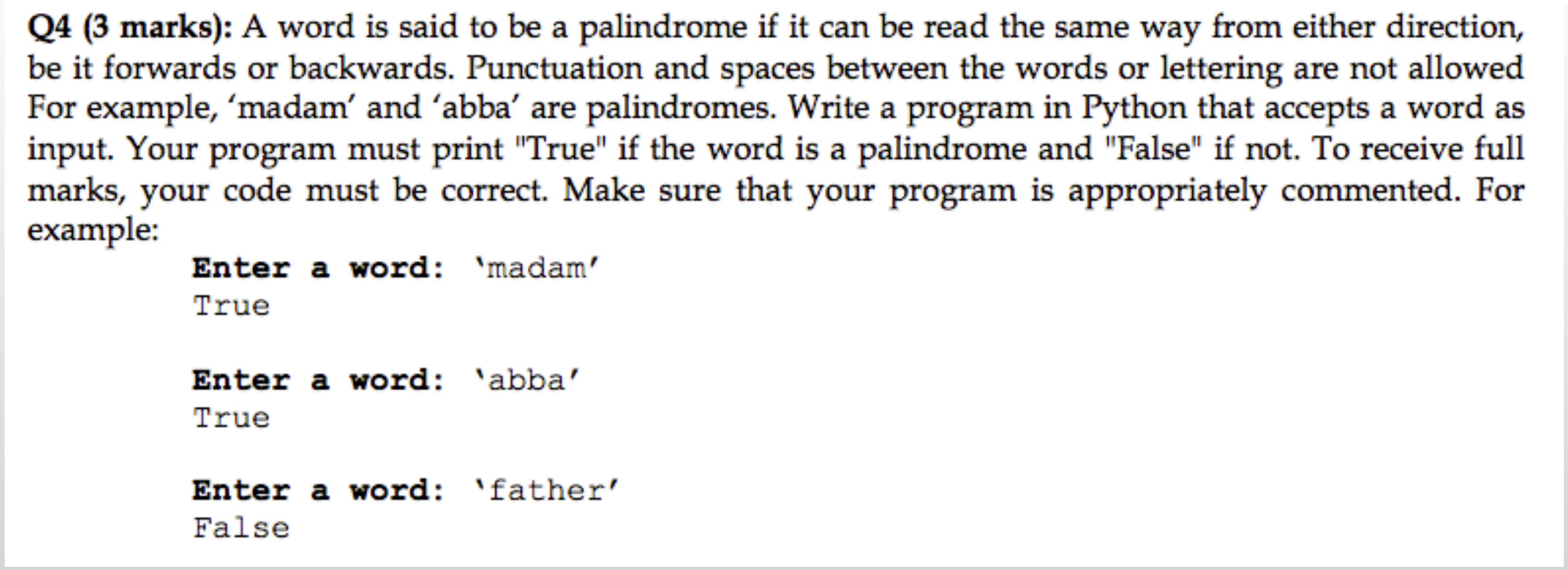
Sample Test



Q3 (2 marks): You are given a program segment in Python as follows:

```
string = 'this is a test case'
tag = 0
for count in range(len(string)):
    if string[count] in "aeiou":
      tag +=1
print(tag)
```

What does this Python program compute? Explain your answer briefly in a sentence or two. What is the final value of the variable *tag*?



Q4 (3 marks): A word is said to be a palindrome if it can be read the same way from either direction, be it forwards or backwards. Punctuation and spaces between the words or lettering are not allowed For example, 'madam' and 'abba' are palindromes. Write a program in Python that accepts a word as input. Your program must print "True" if the word is a palindrome and "False" if not. To receive full marks, your code must be correct. Make sure that your program is appropriately commented. For example:

```
Enter a word: 'madam'
True

Enter a word: 'abba'
True

Enter a word: 'father'
False
```