ISYS90088
Introduction to Application Development

Contd. from Week 4 lectures – for using the range function

Week 5 lectures –nested for, while; formatting

Department of Computing and Information Systems
University of Melbourne
Semester 2 , 2018

Dr Antonette Mendoza

Objectives

- For and nested for statement
- While statement
- Examples
- Formatting and examples

Loops in Python

- Python programming language provides following types of loops to handle looping requirements.

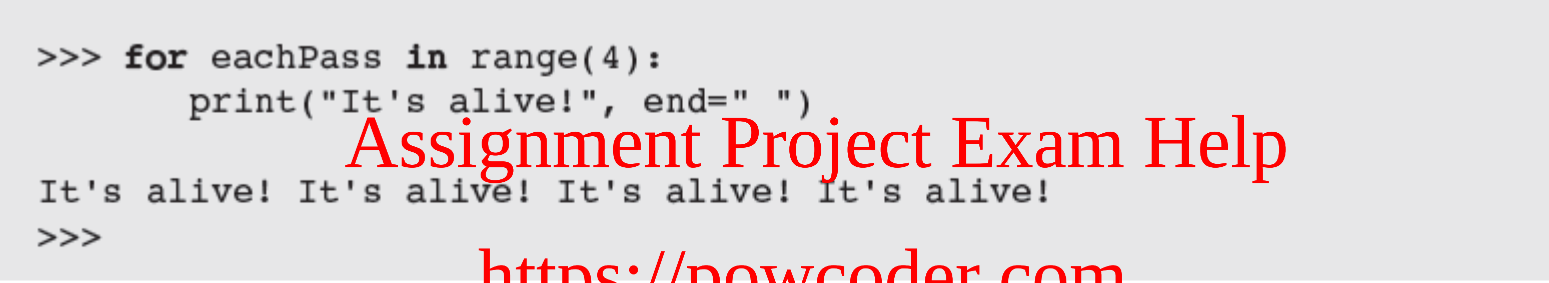
- Types of loops:

  - **for loop:** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

  - **while loop:** Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

  - **nested loops:** can use one or more loop inside any another while, for or while loop.

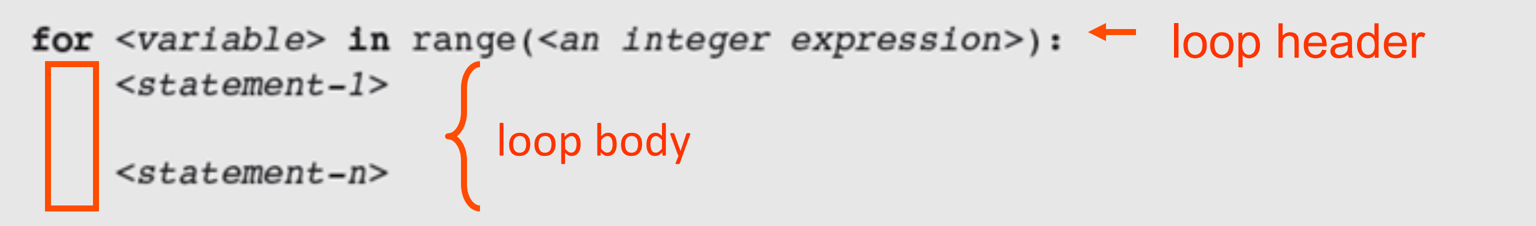
Executing a Statement a Given Number of Times using the **range** function



```
>>> for eachPass in range(4):
        print("It's alive!", end=" ")

It's alive! It's alive! It's alive! It's alive!
>>>
```

- The form of this type of loop is:



```
for <variable> in range(<an integer expression>):
    <statement-1>

    <statement-n>
```

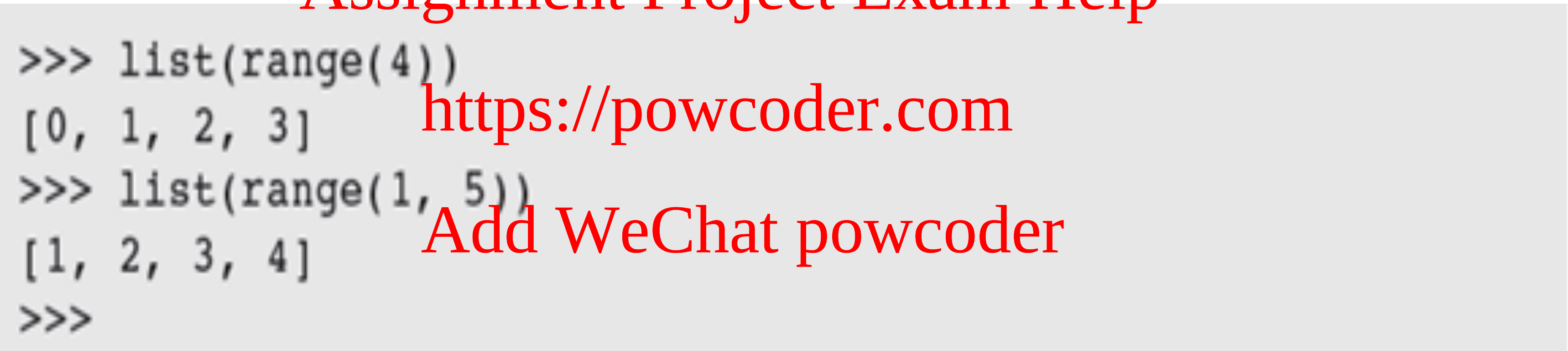
loop header

loop body

statements in body must be indented and aligned in the same column

Traversing the Contents of a Data Sequence

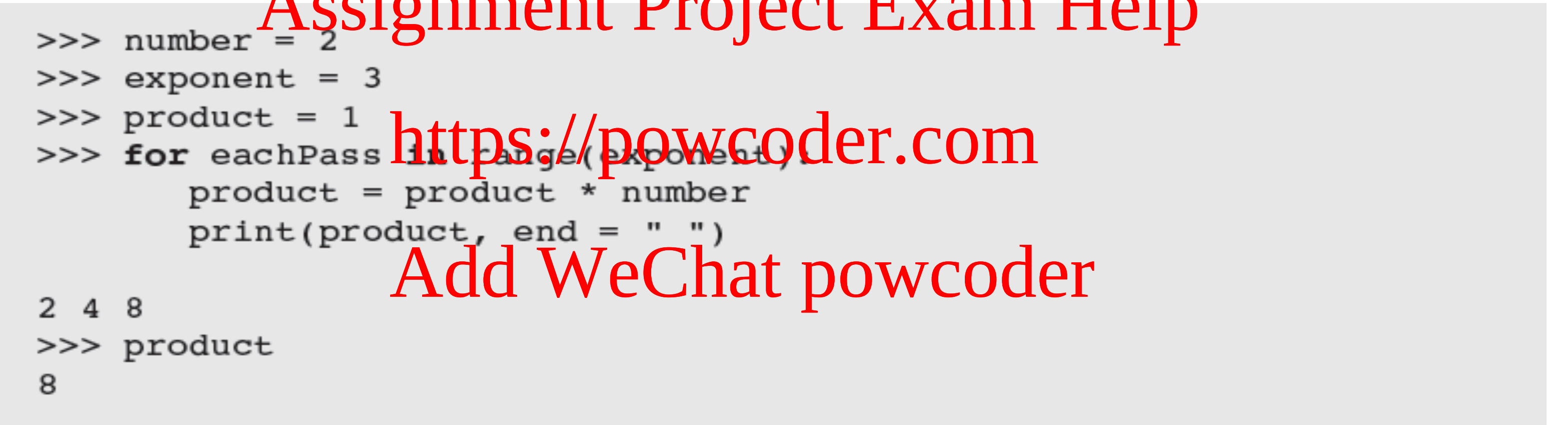
- **`range`** returns a **list**



```
>>> list(range(4))
[0, 1, 2, 3]
>>> list(range(1, 5))
[1, 2, 3, 4]
>>>
```

Executing a Statement a Given Number of Times (continued)

- Example: Loop to compute an exponentiation for a non-negative exponent



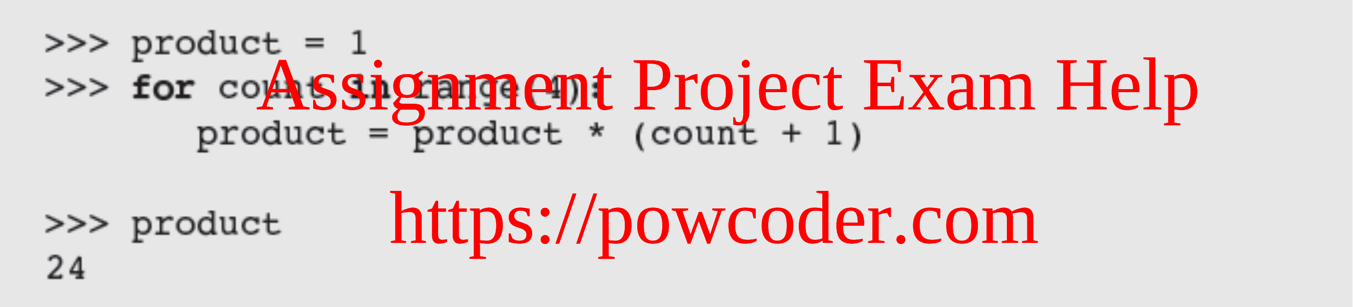
```
>>> number = 2
>>> exponent = 3
>>> product = 1
>>> for eachPass in range(exponent):
       product = product * number
       print(product, end = " ")

2 4 8
>>> product
8
```

- The variable **product** is called an accumulator
- If the exponent were 0, the loop body would not execute and value of **product** would remain as 1

Count-Controlled Loops

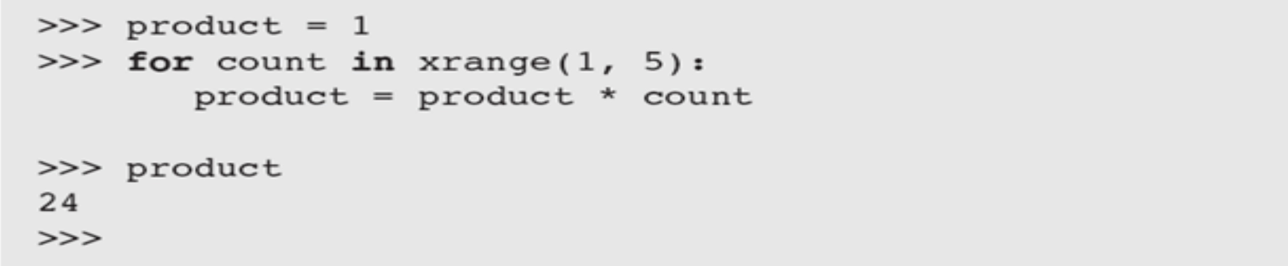
- Loops that count through a range of numbers



```
>>> product = 1
>>> for count in range(4):
        product = product * (count + 1)

>>> product
24
```

- To specify a explicit lower bound:

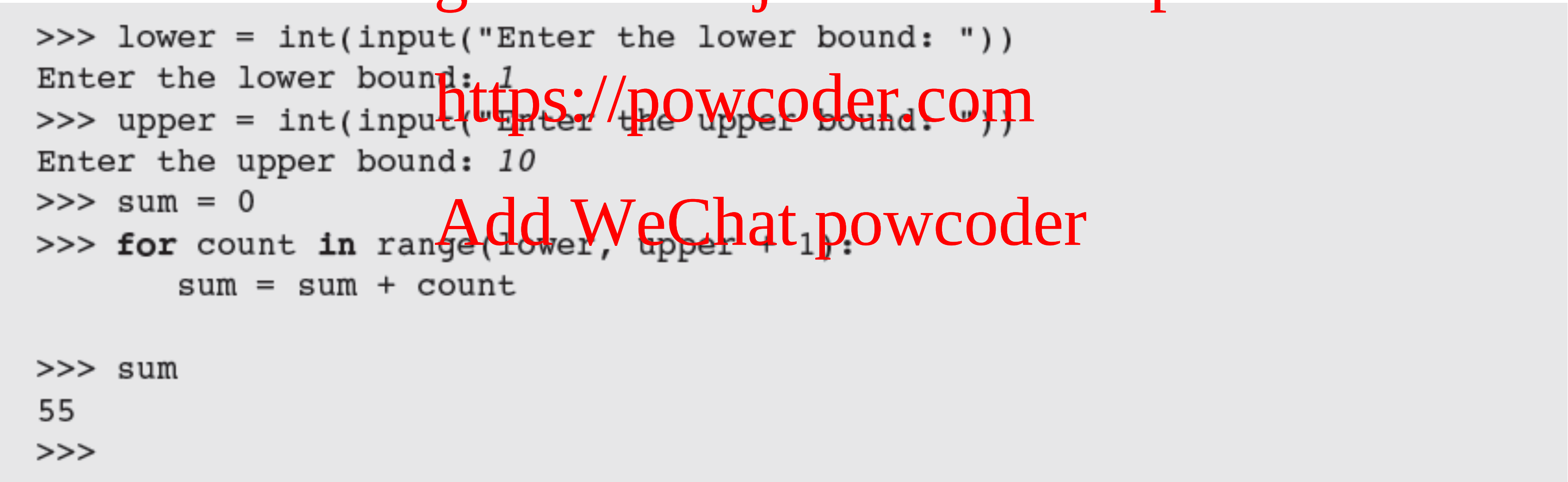


```
>>> product = 1
>>> for count in xrange(1, 5):
        product = product * count

>>> product
24
>>>
```

Count-Controlled Loops (continued)

- Example: bound-delimited **summation**

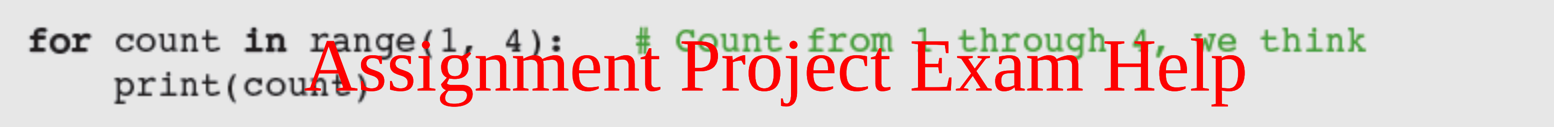


```
>>> lower = int(input("Enter the lower bound: "))
Enter the lower bound: 1
>>> upper = int(input("Enter the upper bound: "))
Enter the upper bound: 10
>>> sum = 0
>>> for count in range(lower, upper + 1):
        sum = sum + count


>>> sum
55
>>>
```

Loop Errors: Off-by-One Error

- Example:



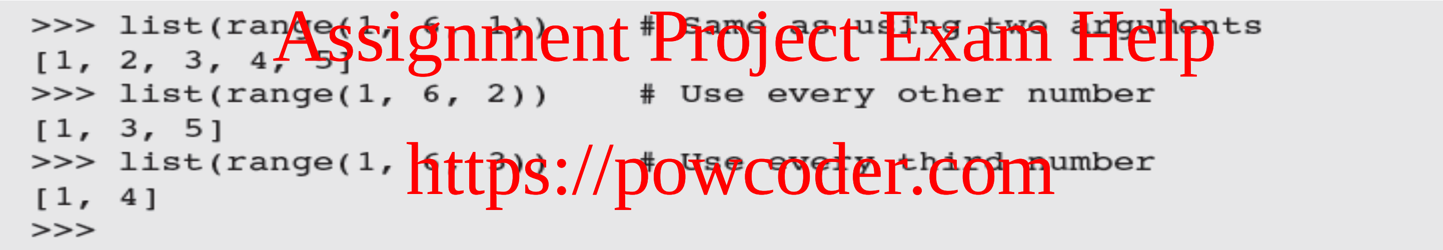
```
for count in range(1, 4):    # Count from 1 through 4, we think
    print(count)
```

Loop actually counts from 1 through 3

- This is not a syntax error, but rather a logic error

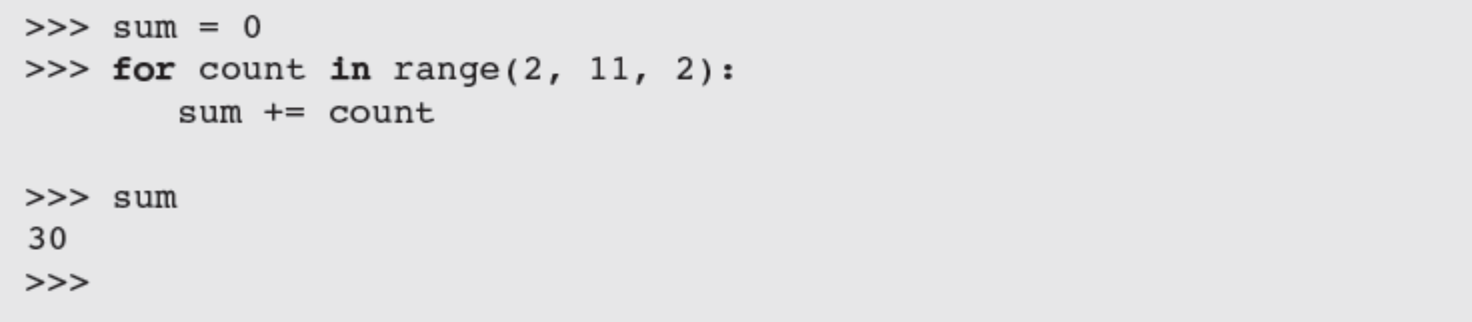
Specifying the Steps in the Range

- **range** expects a third argument that allows you specify a **step value**



```
>>> list(range(1, 6, 1))     # Same as using two arguments
[1, 2, 3, 4, 5]
>>> list(range(1, 6, 2))     # Use every other number
[1, 3, 5]
>>> list(range(1, 6, 3))     # Use every third number
[1, 4]
>>>
```

- Example in a loop:

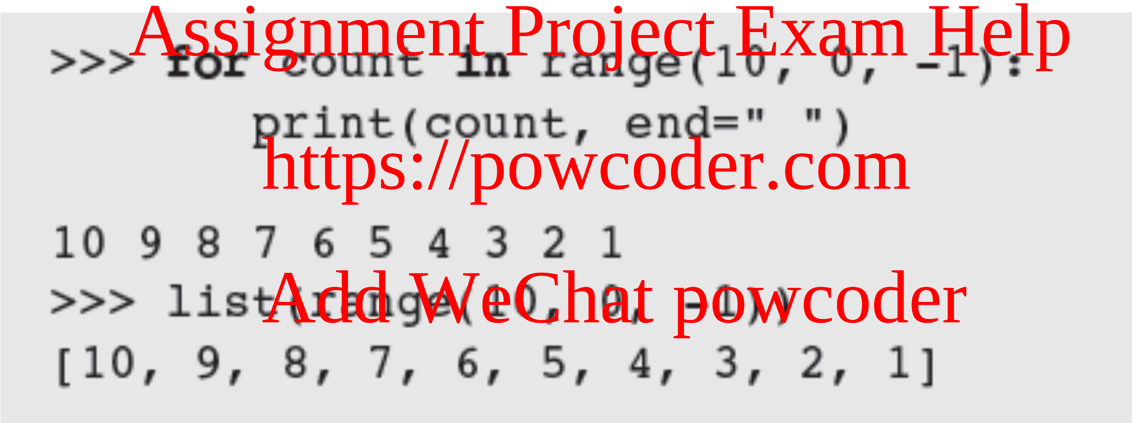


```
>>> sum = 0
>>> for count in range(2, 11, 2):
        sum += count

>>> sum
30
>>>
```

Loops That Count Down

- Example:



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

10 9 8 7 6 5 4 3 2 1
>>> list(range(10, 0, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Quiz

1. Write the output of the following loops:

   a. for count **in** range(5)

      print(count +1, end=" ")

   b. **for** count **in** range(1, 4):

      print(count)

   c. **for** count **in** range(1, 6, 2):

      print(count)

   d. **for** count **in** range(6, 1, -1):

      print(count)

Nested `for` loops

Syntax for nested `for`**:**

```
for iterating_var in sequence:
    for iterating_var in sequence:
            statements(s)
    statements(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 (in 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+1):
    s = “#"
    for y in range(x-1):
        s += symbol
    print (s)
```

Examples: A simple nested `for` loop

Example code that checks whether numbers between 1 and 10 are prime.

## to calculate if a number is prime or not

```
for num in range(1,10):   #to iterate between 1 to 10
    for i in range(2,num):  #to iterate on the factors of the number
        if num%i == 0:        #to determine the factor
            print (num,'is not prime')
            break
    else:
        print (num, 'is a prime number')
```

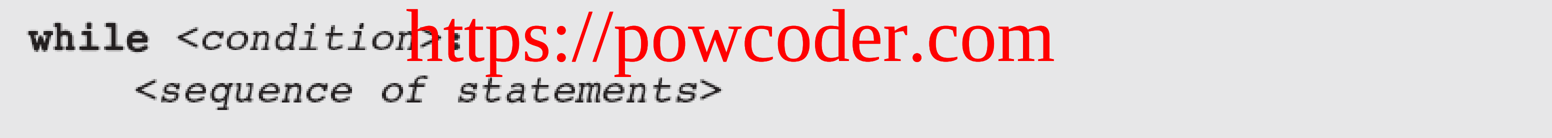
- Try doing this using a while as home work!!!!!

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

Structure and Behavior of a while Loop

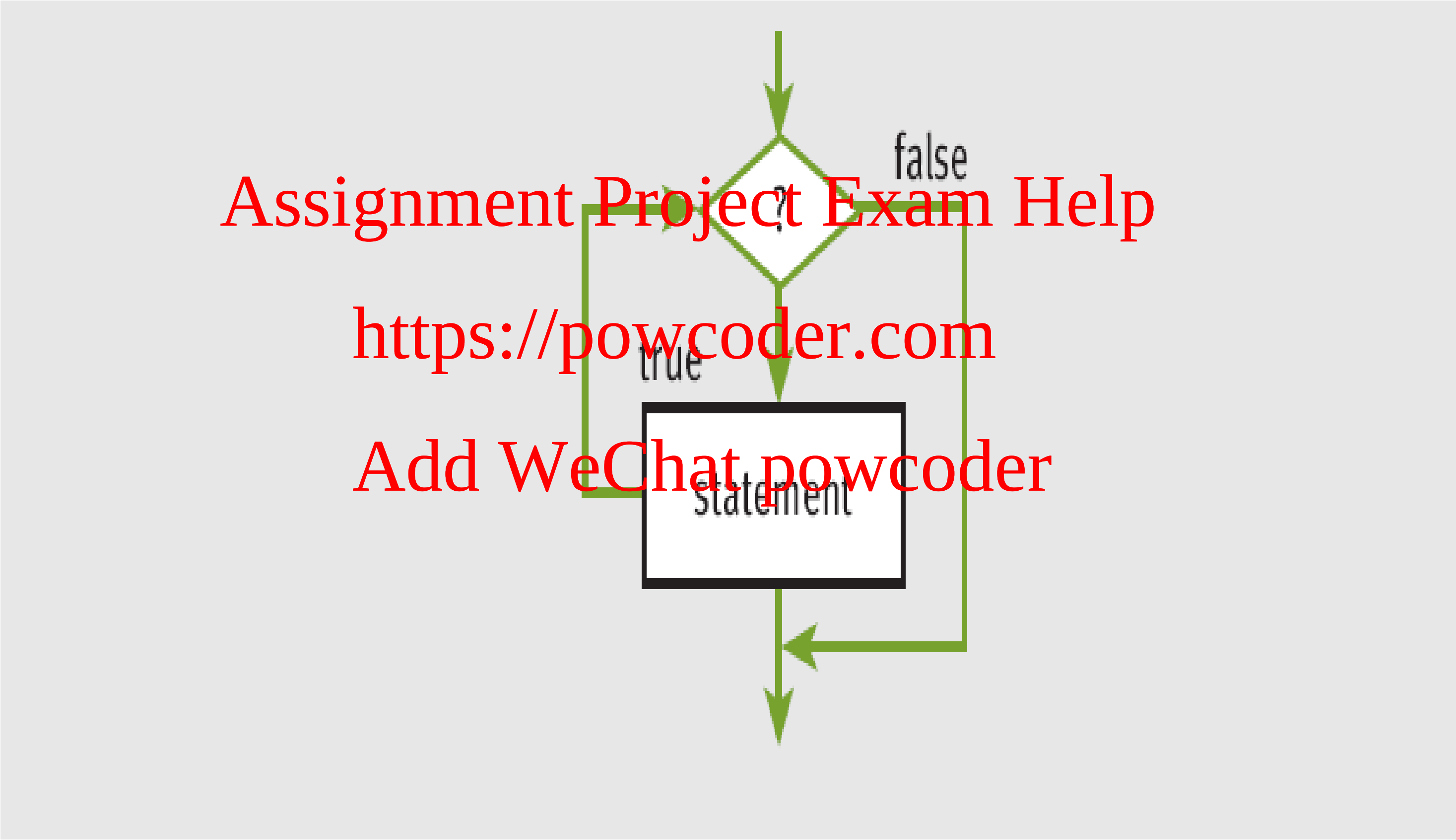
- Conditional iteration requires that condition be tested within loop to determine if it should continue
  - Called **continuation condition**



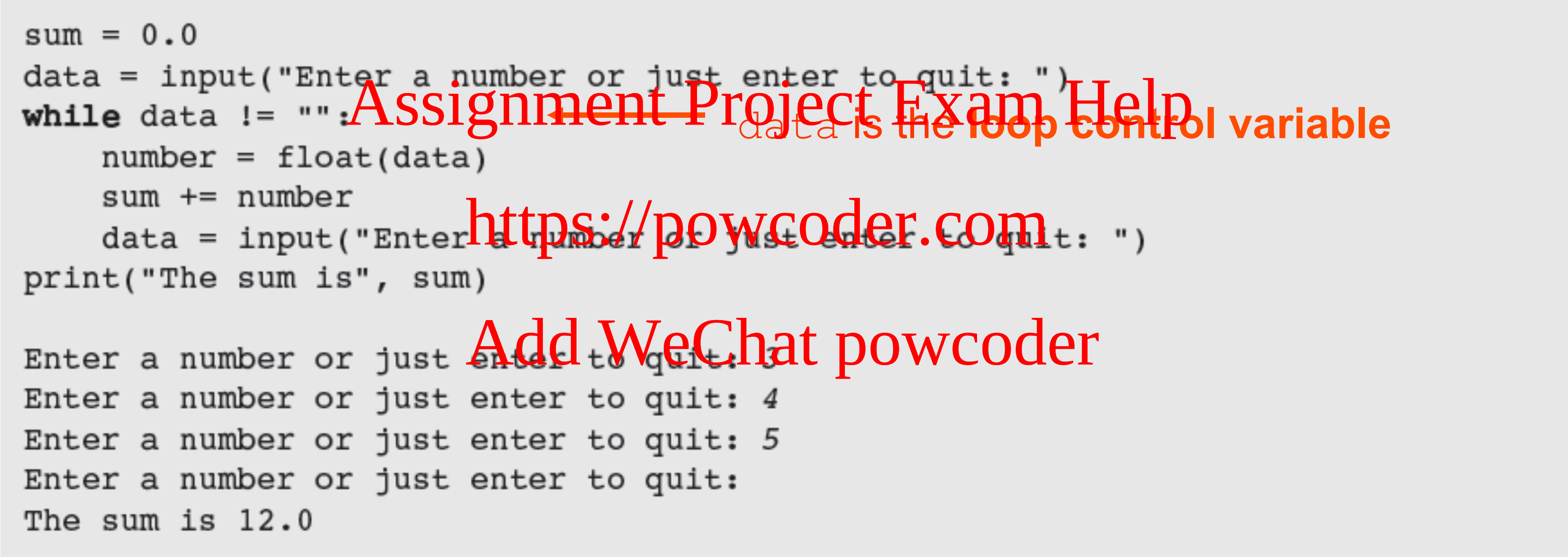
```
while <condition>:
    <sequence of statements>
```

  - 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

Structure and Behavior of a while Loop



Structure and Behavior of a while Loop (continued)

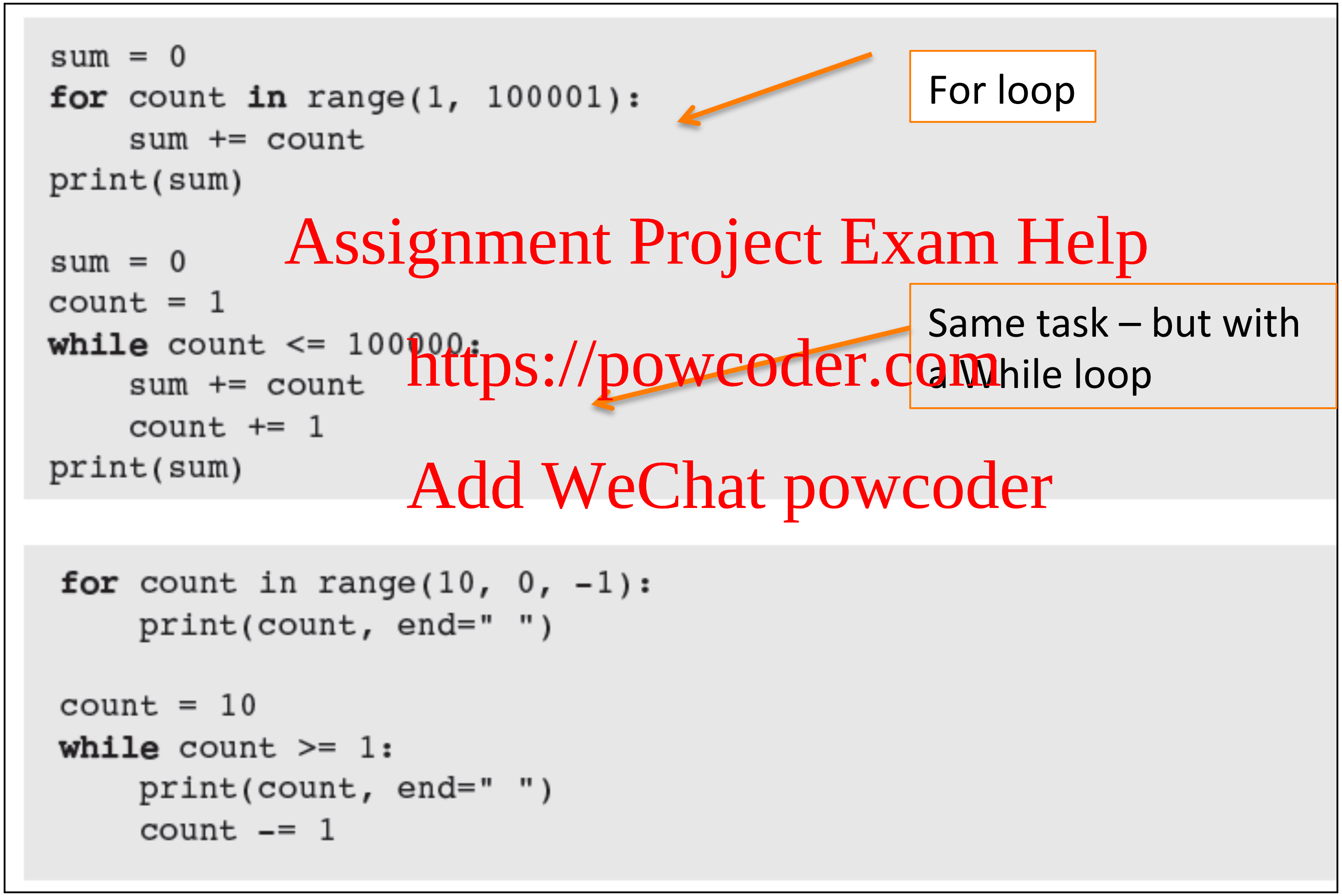


```
sum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    sum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", sum)

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 loop

Same task – but with a While loop

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

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

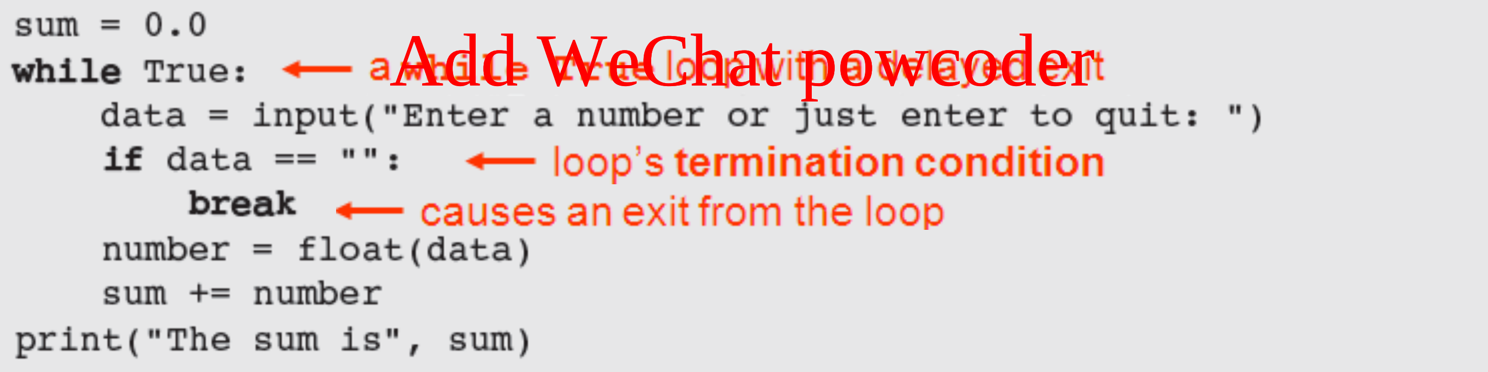
Nested while

Syntax for nested while:

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

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

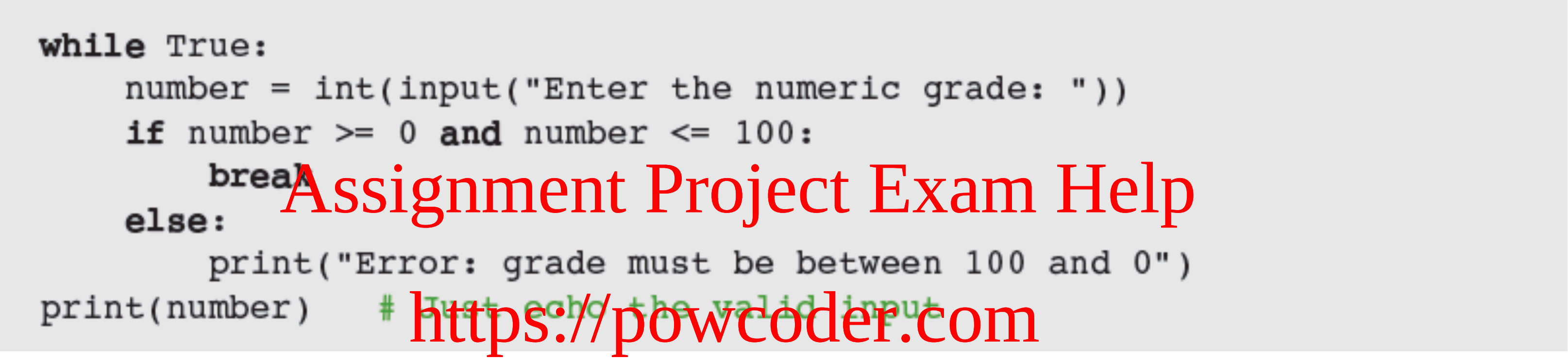
- **while** loop can be complicated to write correctly
  - Possible to simplify its structure and improve its readability



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

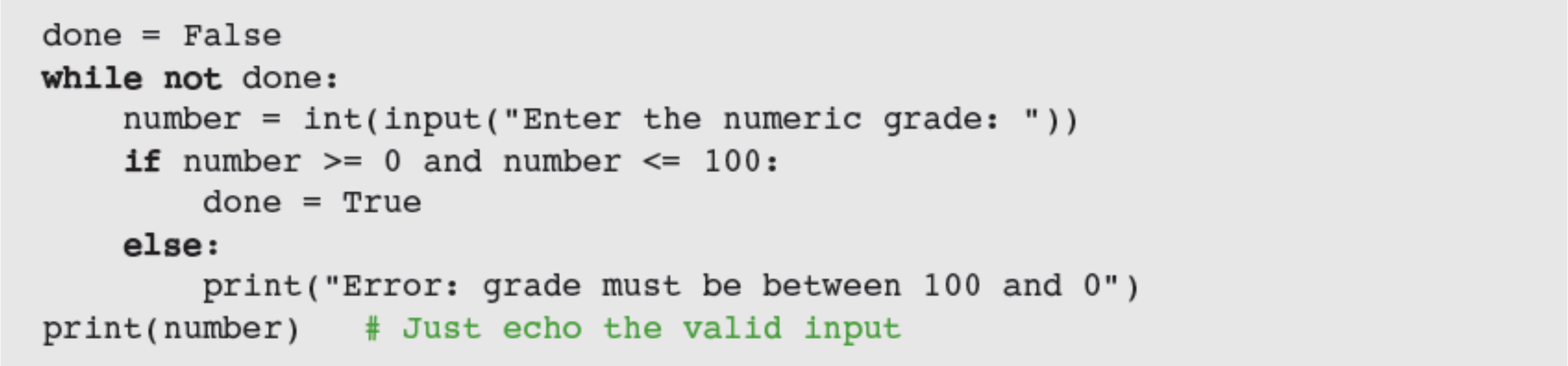
a while True loop with a delayed exit
loop's termination condition
causes an exit from the loop

The `while True` Loop and the `break` Statement (continued)



```
while True:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        break
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```

- Alternative: Use a Boolean variable to control loop



```
done = False
while not done:
    number = int(input("Enter the numeric grade: "))
    if number >= 0 and number <= 100:
        done = True
    else:
        print("Error: grade must be between 100 and 0")
print(number)    # Just echo the valid input
```

break statement

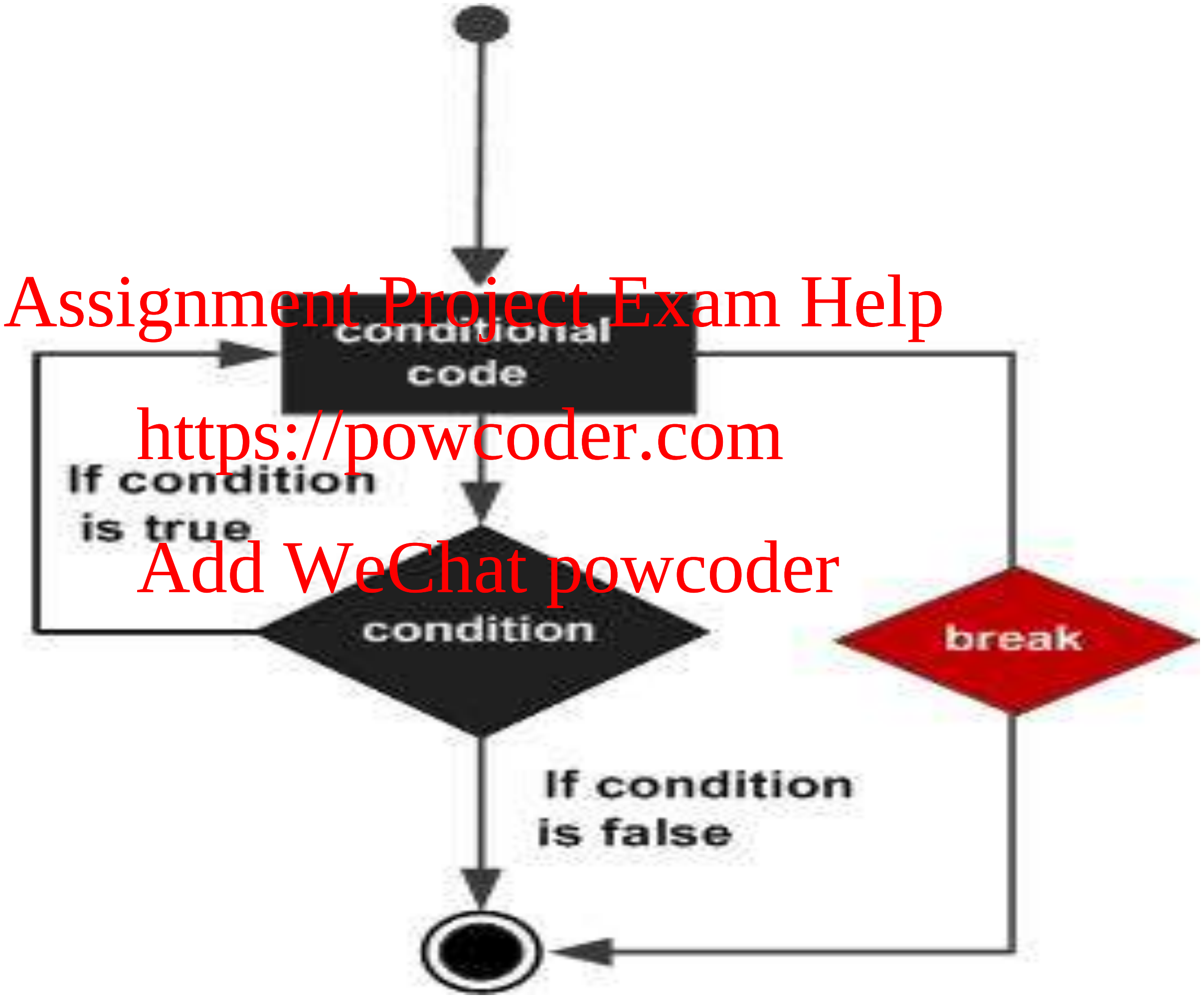
- It terminates the current loop and resumes execution at the next statement
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

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

Syntax:

```
break
```

break statement



Example

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!")
```

When to use : for and while loop

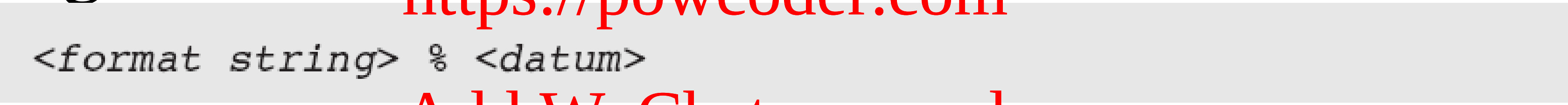
Simplest way to differentiate between the **for** and the **while**:

- 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**.

Formatting for output

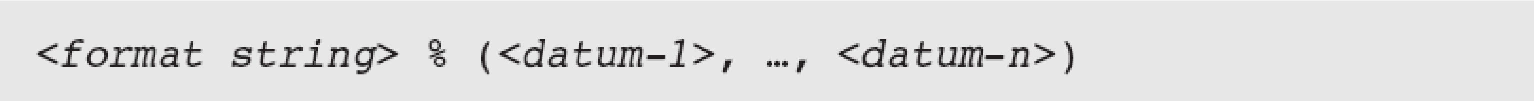
Formatting Text for Output

- Use formatting when we need output that has **tabular format**
- **Field width**: Total number of data characters & additional spaces for a datum in a formatted string



```
<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:

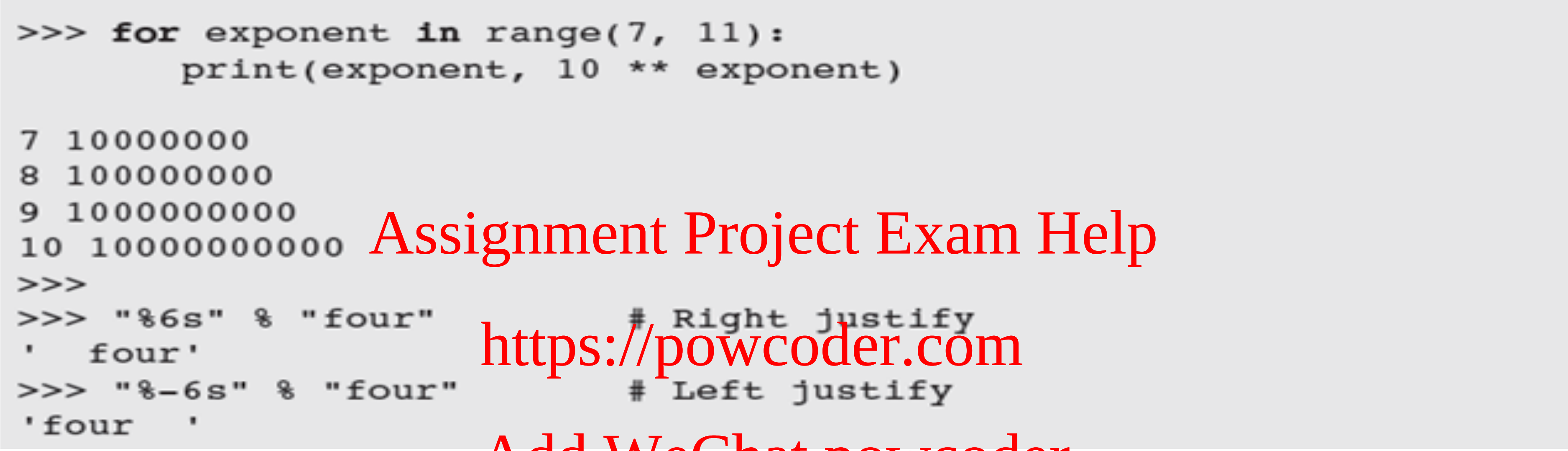


```
<format string> % (<datum-1>, …, <datum-n>)
```

Formatting Text for Output (continued)

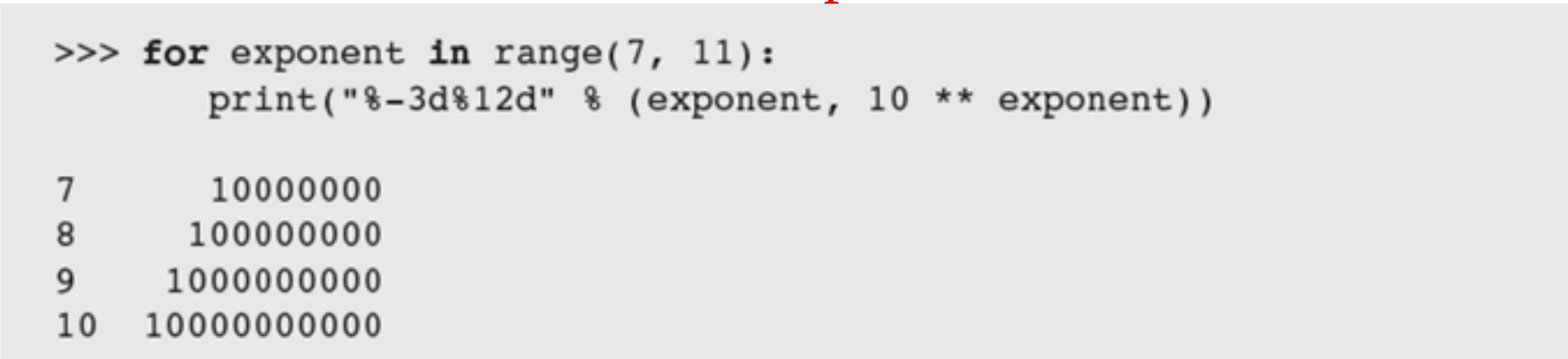
- When the field width is positive, the datum is right justified
- When the field width is negative, the datum is left justified
- If the field width is less that or equal to the datum’s print length in characters, no justification is added.

Examples



```
>>> 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  '
```

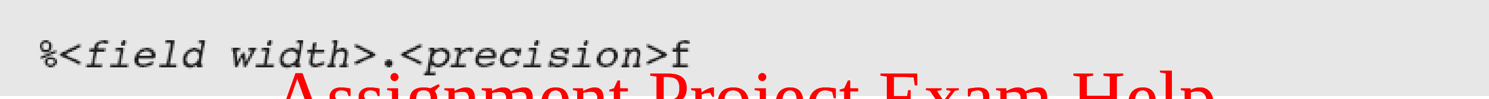


```
>>> for exponent in range(7, 11):
       print("%-3d%12d" % (exponent, 10 ** exponent))

7      10000000
8     100000000
9    1000000000
10  10000000000
```

Formatting Text for Output (continued)

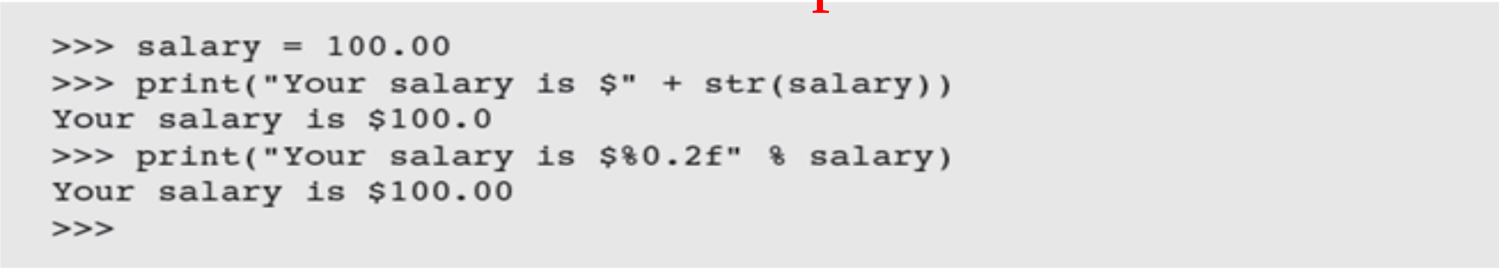
- To format data value of type **float**:



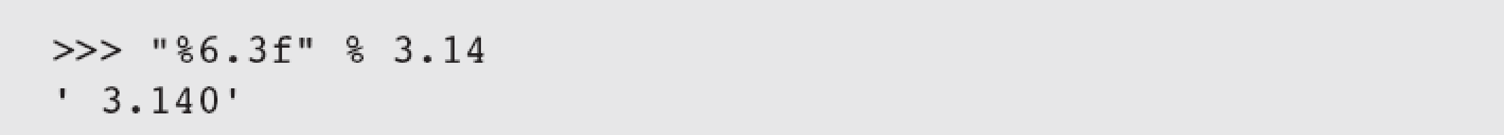
```
%<field width>.<precision>f
```

where .*<precision>* is optional

- Examples:



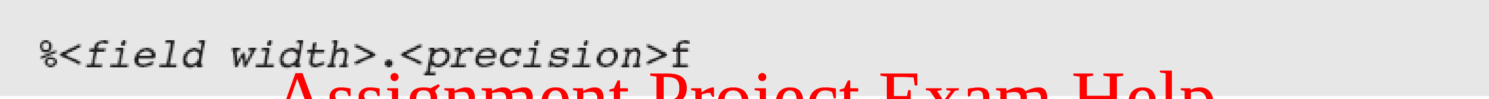
```
>>> salary = 100.00
>>> print("Your salary is $" + str(salary))
Your salary is $100.0
>>> print("Your salary is $%0.2f" % salary)
Your salary is $100.00
>>>
```



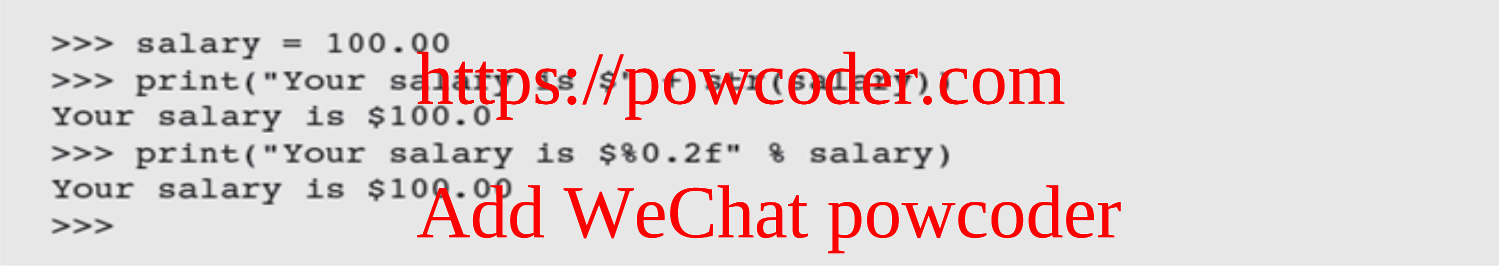
```
>>> "%6.3f" % 3.14
' 3.140'
```

Formatting Text for Output (continued)

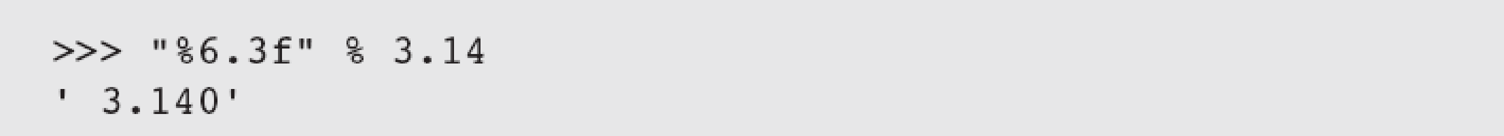
- Examples:



```
%<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: Quiz

```
>>>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'))
```

Example : formatting quiz

- 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))
```

(check out many more examples on LMS)

Formatting multiple values

Syntax:

```
print (<format 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)
```

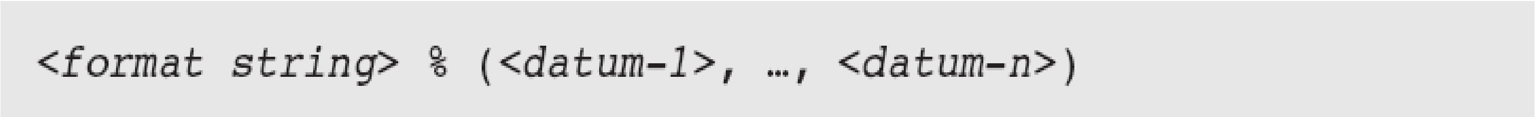
Formatting numbers and strings

Note:

- specifying a minimum field width - is the minimum number of spaces that should be used to display a value
- the field width specifies the number of spaces reserved on the screen for the value.
- if the value is shorter than the field width, it is displayed and will be right justified (filled with spaces)
- if the value is too large to fit in the specified field width, the field is automatically enlarged to accommodate it.

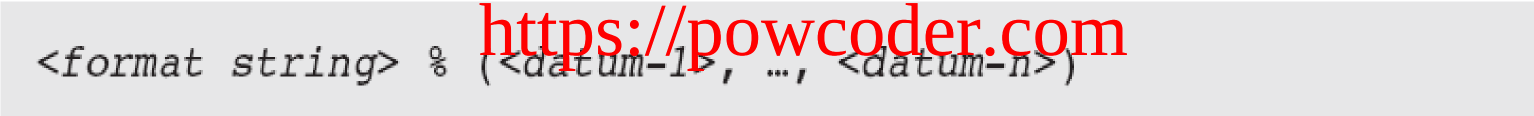
Formatting: examples (new styling (vs) old style of formatting

- Old style syntax:



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

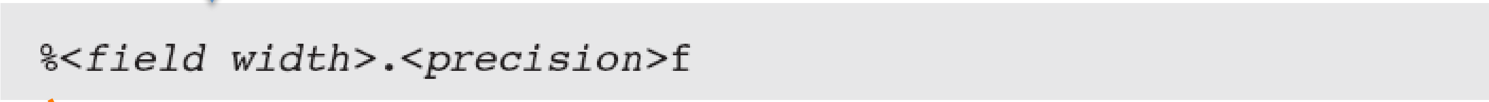
- New formatting style syntax in general:



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

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

Old:



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

New:

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

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 (new styling)

#Examples for formatting using the format()

# using <, >, ^ and a filler

```
>>>print('{:_<10}'.format('test'))
#left
>>>print('{:_^10}'.format('test'))
#centered
>>>print('{:_>10}'.format('test'))
#right
>>>print('{:*>10}'.format('test'))
# * as a filler
```

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))
```

```
Output:
he number contains 10 digits
The digits sum to 100
```

Side notes – lots of them to check out! (see uploaded on the LMS. There are a few rule changes when you use the format() – new style of formatting

- You may use the old or the new style to format

Formatting: some more examples to try out!

# 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]
  print('{d[4]} {d[5]}'.format(d=data))
```