From Last week

Slicing for Substrings

- Python's subscript operator can be used to obtain a substring through a process called **slicing**
  - Place a colon (**:**) in the subscript; an integer value can appear on either side of the colon
- For example, the following code accesses the substring of the string **"the example is on slicing"** starting at index **2**, up to (but not including) index **10**.

```
>>>s = "the example is on slicing"
>>>print (s[2:10])
e exampl
```

Accessing Substrings (Slicing)

- The notation **2:10** is known as a slice.
- Remember that a slice starts at the first index but finishes one before the end index. This is consistent with indexing: indexing also starts from zero and goes up to one before the length of the string.
- You can see this by slicing with the value of **len**:

```
>>>s = "the example is on slicing"
>>print (len(s))
>>print (s[0:len(s)])
```

More on Slicing: example

- You can also slice with negative indices. The same basic rule of starting from the start index and stopping one before the end index applies:

```
>>>s = ”testing slicing 101"
>>>print(s[4:-7])
>>>print(s[-7:-1])
>>>print(s[-6:len(s)])
```

Solution:
????

Example 7

Changing the Step Size and Direction

- You can specify a third number which indicates how much to step through the list by.
- For example, if you want every second element you can do this:

```
>>>s = "abcdef"
>>>print(s[::2])
```

- Or you can specify a range with a step:

```
>>>s = "abcdef"
>>>print(s[0:3:2])
```

(Example 8)

Changing the Step Size and Direction

- If this third number is -1 it changes the direction you are slicing in:

```
>>>s = "abcdef"
>>>print(s[2::-1])
>>>print(s[2:0:-1])
>>>print(s[-4:-6:-1])
cba
cb
cb
```

- Note that the direction of the indices must be changed also. If there is nothing in between the indices, an empty string is returned (unlike indexing beyond the ends of the string, which led to an error):

```
>>>s = "abcdef"
>>>print(s[0:2:-1])
```

ISYS90088
Introduction to Application Development

Contd. from Week 4 lectures – String methods

Week 5 lectures – For statement, formatting;

Department of Computing and Information Systems
University of Melbourne
Semester 2 , 2017

Dr Thomas Christy

Objectives (continued)

- String sequences and library functions to manipulate with strings
- Examples
- For statement
- Formatting

Testing for a Substring with the in Operator

- When used with strings, the left operand of **in** is a target substring and the right operand is the string to be searched
  - Returns **True** if target string is somewhere in search string, or **False** otherwise

- Example:

```
>>>list = [“ant”, “dog”, “cat”, “rat”, “horse”]
>>>”ant” in list
True
```

String Methods

- Python includes a set of string operations called **methods** that make tasks like counting the words in a single sentence easy
- dir(str) or help(str) will give you the entire list of string methods or library functions.
- A method behaves like a function, but has a slightly different syntax
  - A method is always called with a given data value called an **object**

```
<an object>.<method name>(<argument-1>, …, <argument-n>)
```

String Methods (continued)

- Methods can expect arguments and return values
- A method knows about the internal state of the object with which it is called
- In Python, all data values are objects

String Methods (continued)

STRING METHOD	WHAT IT DOES
`s.center(width)`	Returns a copy of `s` centered within the given number of columns.
`s.count(sub [, start [, end]])`	Returns the number of non-overlapping occurrences of substring `sub` in `s`. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.endswith(sub)`	Returns `True` if `s` ends with `sub` or `False` otherwise.
`s.find(sub [, start [, end]])`	Returns the lowest index in `s` where substring `sub` is found. Optional arguments `start` and `end` are interpreted as in slice notation.
`s.isalpha()`	Returns `True` if `s` contains only letters or `False` otherwise.
`s.isdigit()`	Returns `True` if `s` contains only digits or `False` otherwise.

String Methods (continued)

STRING METHOD	WHAT IT DOES
`s.join(sequence)`	Returns a string that is the concatenation of the strings in the sequence. The separator between elements is `s`.
`s.lower()`	Returns a copy of `s` converted to lowercase.
`s.replace(old, new [, count])`	Returns a copy of `s` with all occurrences of substring `old` replaced by `new`. If the optional argument `count` is given, only the first `count` occurrences are replaced.
`s.split([sep])`	Returns a list of the words in `s`, using `sep` as the delimiter string. If `sep` is not specified, any whitespace string is a separator.
`s.startswith(sub)`	Returns `True` if `s` starts with `sub` or `False` otherwise.
`s.strip([aString])`	Returns a copy of `s` with leading and trailing whitespace (tabs, spaces, newlines) removed. If `aString` is given, remove characters in `aString` instead.
`s.upper()`	Returns a copy of `s` converted to uppercase.

String Methods (continued)

```
>>> s = "Hi there!"
>>> len(s)
9
>>> s.center(11)
' Hi there! '
>>> s.count('e')
2
>>> s.endswith("there!")
True
>>> s.startswith("Hi")
True
>>> s.find('the')
3
>>> s.isalpha()
False
>>> 'abc'.isalpha()
True
>>> "326".isdigit()
True
>>> words = s.split()
>>> words
['Hi', 'there!']
>>> "".join(words)
'Hithere!'
```

String Methods (continued)

```
>>> " ".join(words)
'Hi there!'
>>> s.lower()
'hi there!'
>>> s.upper()
'HI THERE!'
>>> s.replace('i', 'o')
'Ho there!'
>>> " Hi there! ".strip()
'Hi there!'
>>>
```

An example for split – an intro to make things interesting - list and tuple

Example 1:

```
>>>s= ‘this is an example’
>>>listofitems = s.split()
>>>print(listofitems)
```

Example 2:

```
>>>s = '2 3 4'
>>>(a, b, c) = s.split()
>>>print(a)
>>>print(b)
>>>print(c)
example(1)
```

Example : program – try this one!

Imagine you are writing a title for a business report in your organization. The title should be not more that 10 words. Write a program that accepts from the user the title and :

i. Makes sure the title is in upper case. If its not, your program converts the title to all upper case;
ii. counts the number of words in the title. If the number of words is >10, your program will inform the user that the title must be not more that 10 words.

(example 9)

Example : fill in the XXX !

```
text = input("enter a sentence :")
If XXX:
    text = text.upper()
print('your title in uppercase is', text)
listofwords =XXX
if XXX > 10:
    print("your title must not exceed 10 words")
else:
    print ("there are", XXX, "words in this title")
```

Loops in Python

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

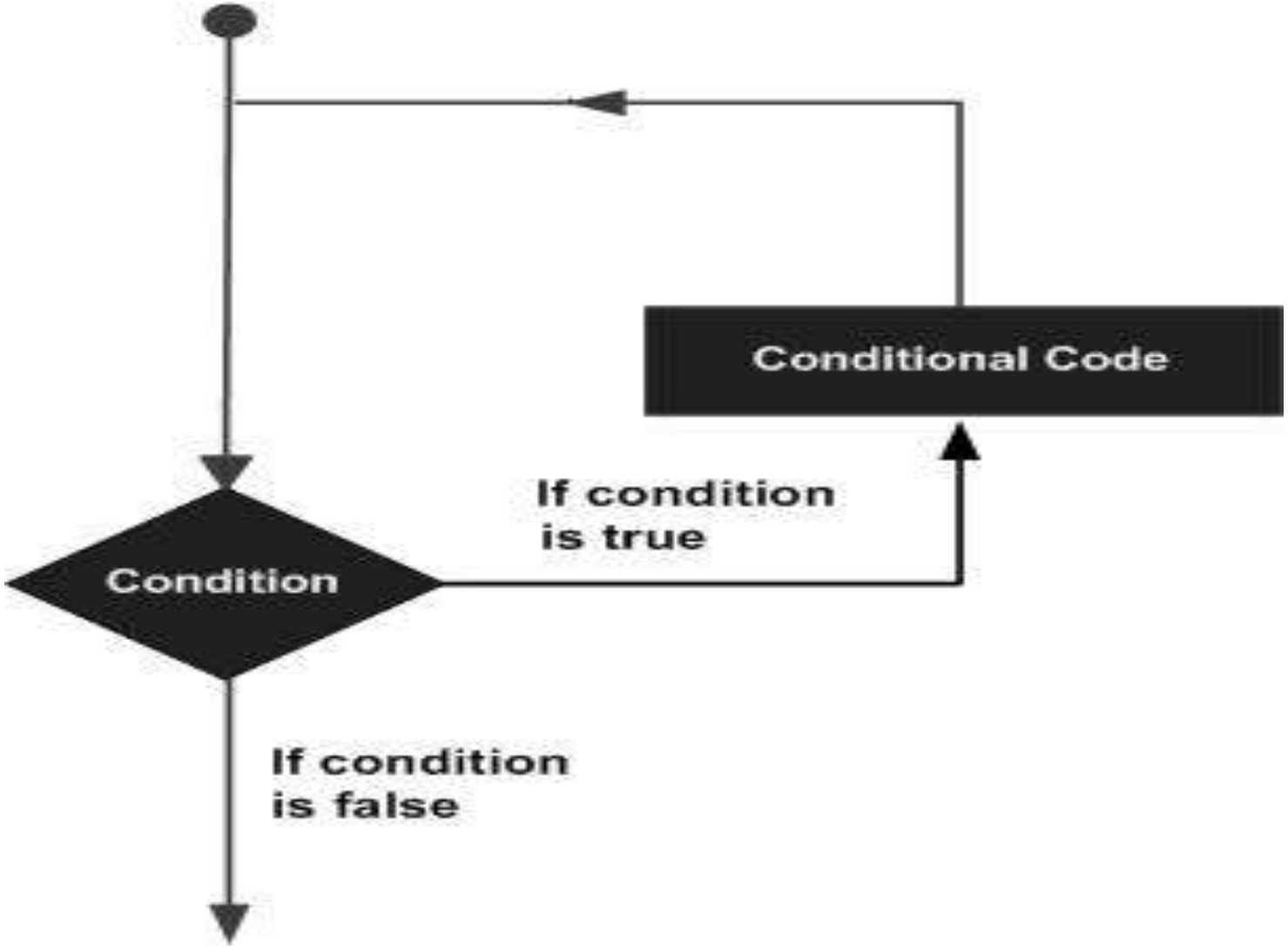
- Types of loops:

  - **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 do..while loop.

Loops


Conditional Code
If condition
is true
Condition
If condition
is false

The **for** Loop

- Repetition statements (or **loops**) repeat an action
- Each repetition of action is known as **pass** or **iteration**
- Python's **for** loop is the control statement that supports definite iteration
- A **for** loop helps with control, counting and repetition

For loop

- It has the ability to iterate over the items of any sequence, such as a list or a string.

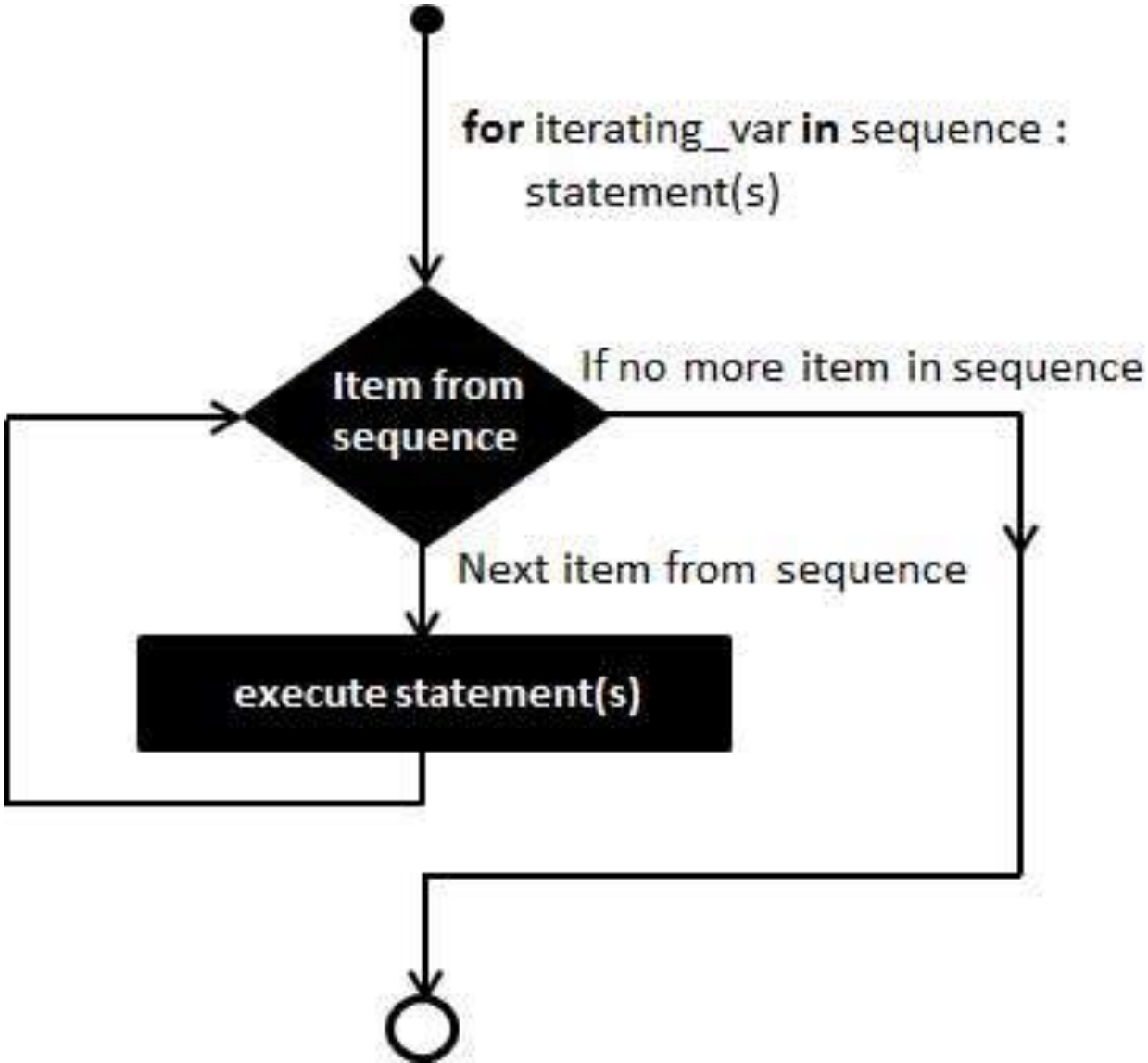
Syntax:

for iterating_var **in** sequence:

statements(s)

- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable iterating_var. Next, the statements block is executed.
- Each item in the list is assigned to iterating_var, and the statement(s) block is executed until the entire sequence is exhausted.

for loop


for iterating_var in sequence :
statement(s)
Item from sequence
If no more item in sequence
Next item from sequence
execute statement(s)

Examples of **for** loops: using list and strings

```
#example 1 -this is one way of doing it
for num in [1, 2, 3, 4, 5]:
    print (num)

#example 2- this is another way of doing it using lists
listofitems = [1, 2, 3, 4, 5]:
for num in listofitems:
    print (num)

# Example using strings
for letter in 'Python':
  print ('Current Letter :', letter)

# Second Example using strings
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:
   print ('Current fruit :', fruit)
```

Traversing the Contents of a Data Sequence

- Strings are also sequences of characters
- Values in a sequence can be visited with a **`for`** loop:

- Example: **character** is called the target variable- that takes a value of each loop iteration

```
for <variable> in <sequence>:
   <do something with variable>
```

```
>>> for character in "Hi there!":
        print(character, end = " ")

H i   t h e r e !
>>>
```

Definite Iteration: The **for** Loop

- Repetition statements (or **loops**) repeat an action
- Each repetition of action is known as **pass** or **iteration**
- Two types of loops
  - Those that repeat action a predefined number of times (**definite iteration**)
  - Those that perform action until program determines it needs to stop (**indefinite iteration**)

Executing a Statement a Given Number of Times

- Python's **for** loop is the control statement that most easily supports definite iteration

```
>>> 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
```

- 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
>>>
```

Augmented Assignment

- **Augmented assignment operations**:

```
a = 17
s = "hi"

a += 3            # Equivalent to a = a + 3
a -= 3            # Equivalent to a = a - 3
a *= 3            # Equivalent to a = a * 3
a /= 3            # Equivalent to a = a / 3
a %= 3            # Equivalent to a = a % 3
s += " there"     # Equivalent to s = s + " there"
```

- Format:

```
<variable> <operator>= <expression>
```

Equivalent to:

```
<variable> = <variable> <operator> <expression>
```

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

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]
>>>
```

- Strings are also sequences of characters
- Values in a sequence can be visited with a **`for`** loop:

```
for <variable> in <sequence>:
   <do something with variable>
```

- Example:

```
>>> for character in "Hi there!":
        print(character, end = " ")

H i   t h e r e !
>>>
```

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]
```

Enumerate loop index

- Example:

```
text = 'Simple'
for index, char_in_text in enumerate(text):
    print('{} is at index {}'.format(char_in_text, index))
print('The end')
```

```
S is at index 0
i is at index 1
m is at index 2
p is at index 3
l is at index 4
e is at index 5
The end
```

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

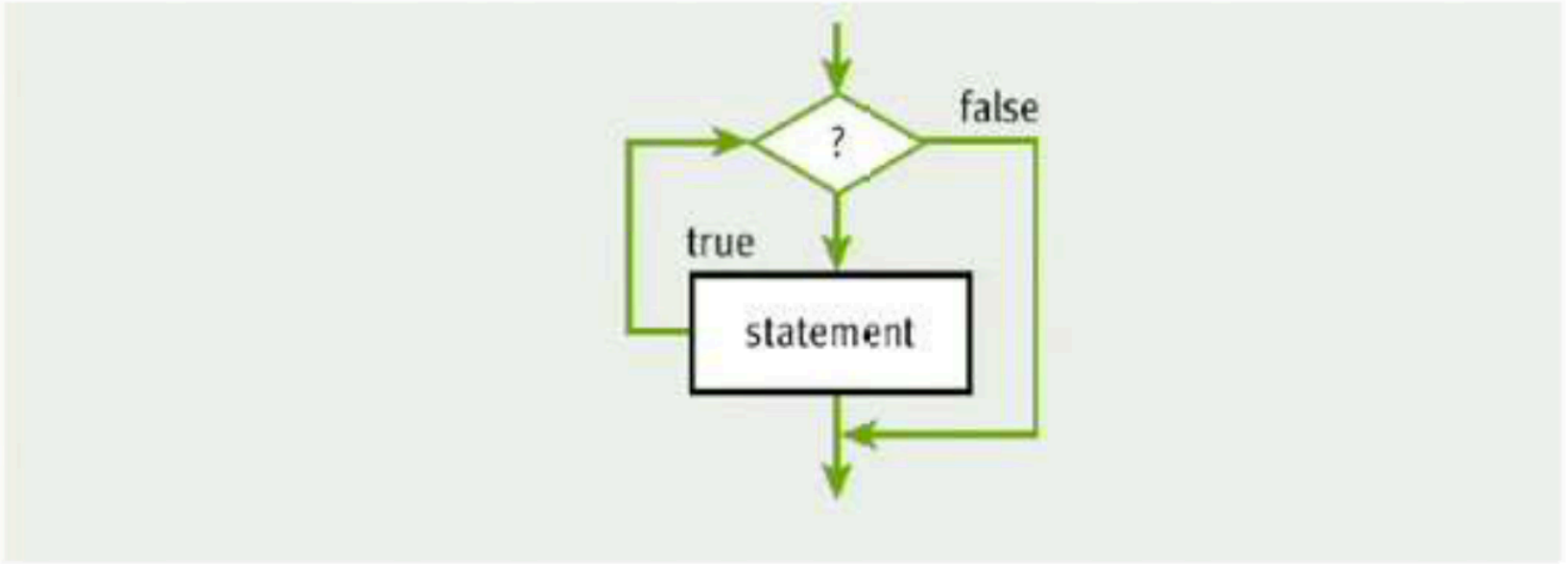
The Structure and Behavior of a `while` Loop

- 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.

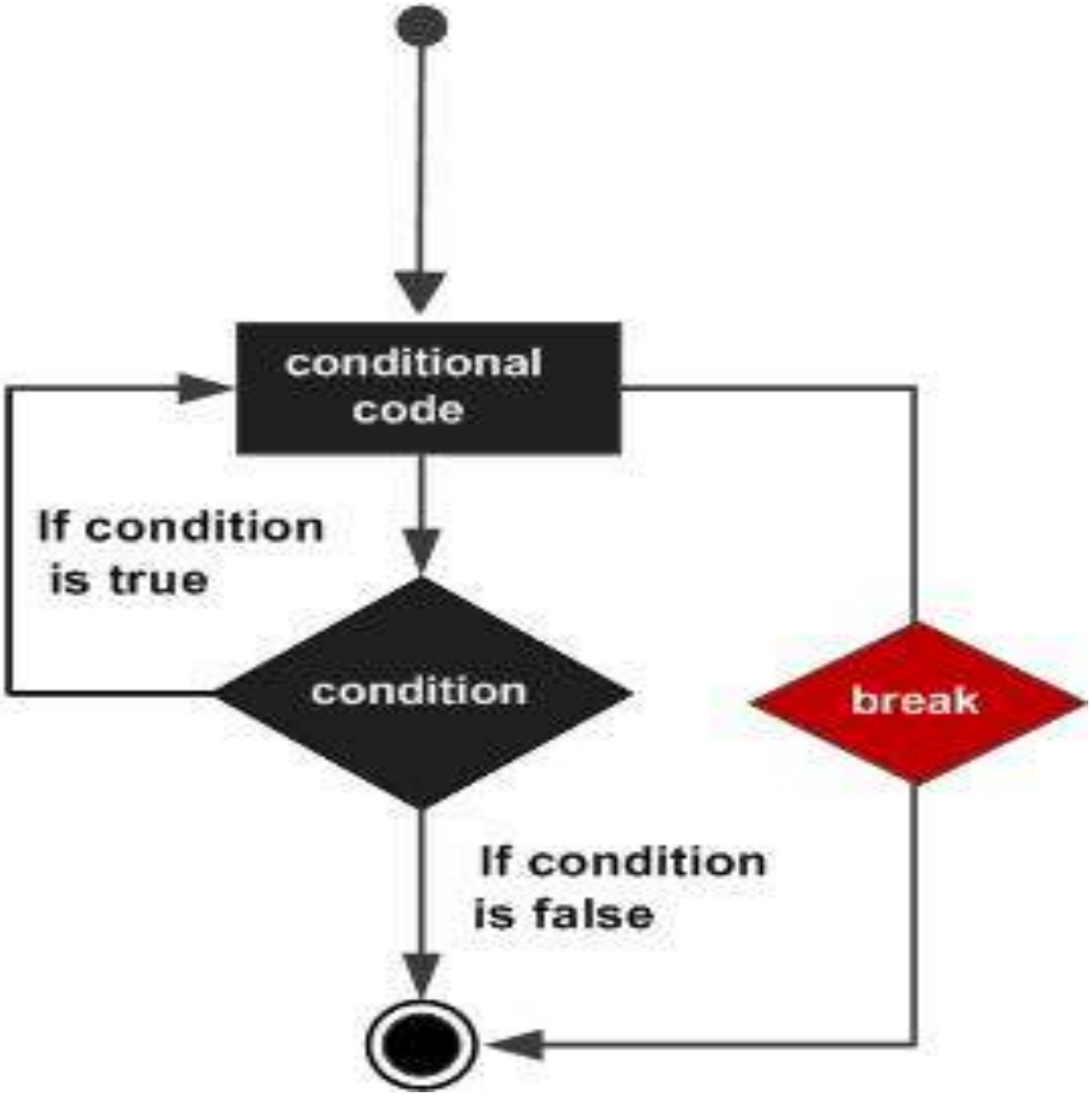
- The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

- 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



continue statement

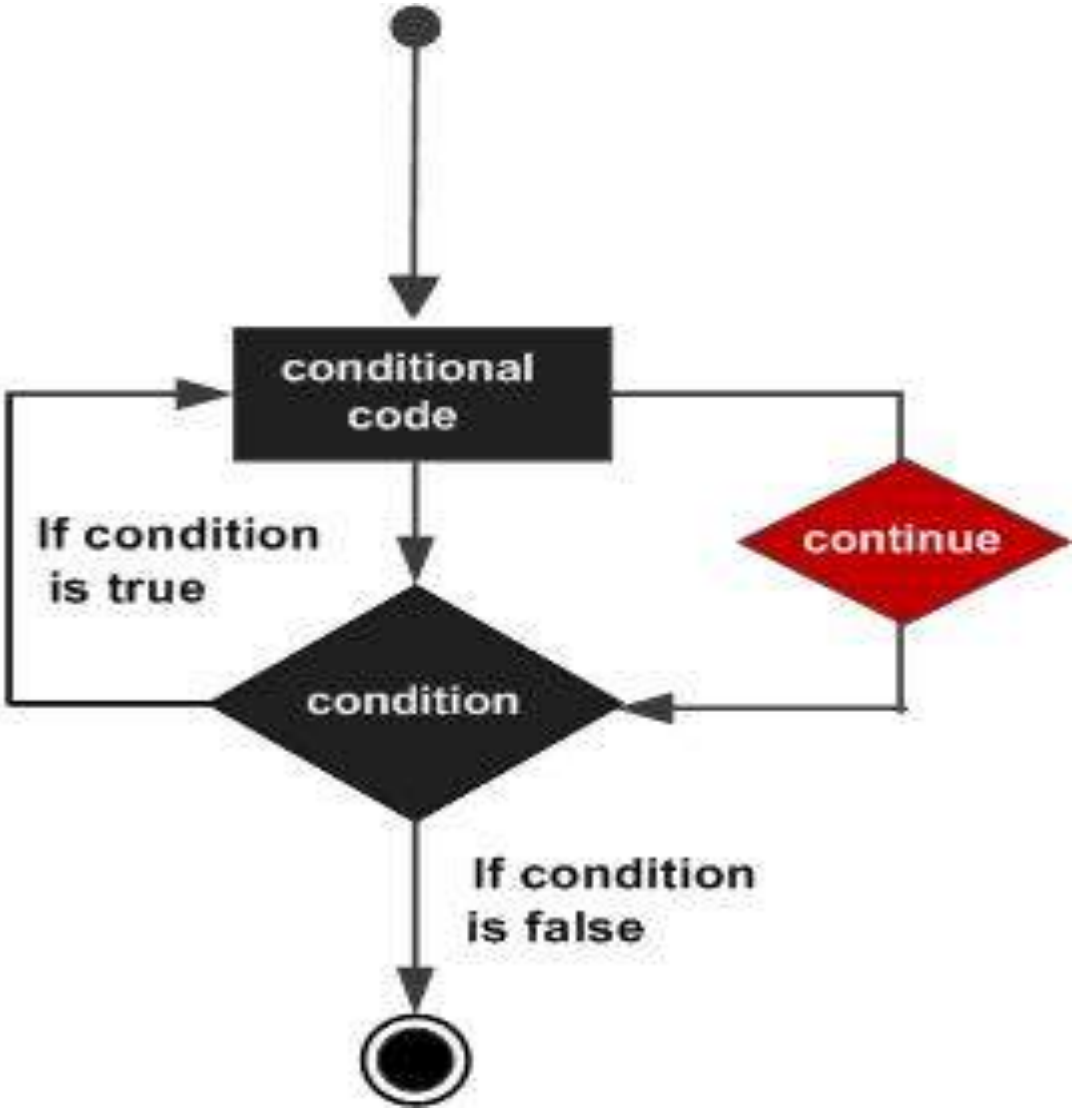
- It returns the control to the beginning of the while loop.. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

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

Syntax:

>>> continue

continue statement



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:

```
# 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))
    startBalance = endBalance
    totalInterest += interest
```

list

Description

- The method list() takes sequence types and converts them to lists. This is used to convert a given tuple into list.

- Note: Tuple are very similar to lists with only difference that element values of a tuple can not be changed and tuple elements are put between parentheses instead of square bracket.

Syntax

- Following is the syntax for list() method:

- list(seq)
- Parameter is seq -- This is a tuple to be converted into list.

- Return Value - This method returns the list.