Execution Control

Kenn H. Kim, Ph. D.

College of Business
Clemson University

Execution Control Structures

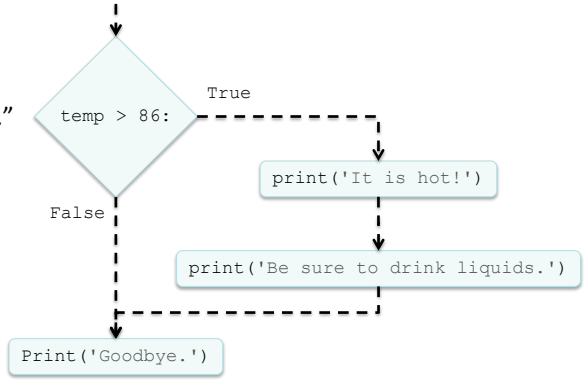
- Conditional Structures
- Iteration Patterns, Part I
- Two-Dimensional Lists
- `while` Loop
- Iteration Patterns, Part II

One-Way `if` Statement

```
if <condition>:
    <indented code block>
<non-indented statement>
```

```
if temp > 86:
    print('It is hot!')
    print('Be sure to drink liquids.')
print('Goodbye.')
```

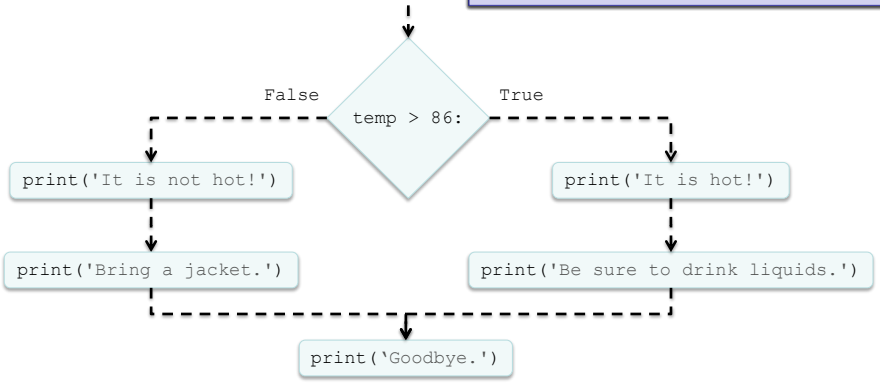
"The value of `temp` is 50."



Two-Way `if` Statement

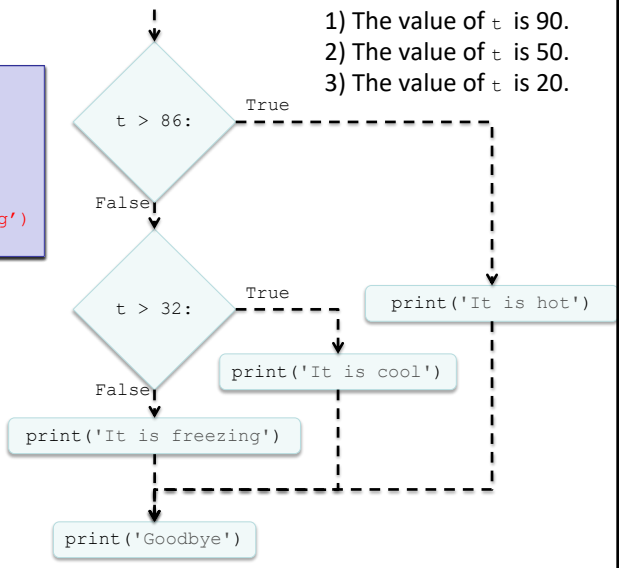
```
if <condition>:
    <indented code block 1>
else:
    <indented code block 2>
<non-indented statement>
```

```
if temp > 86:
    print('It is hot!')
    print('Be sure to drink liquids.')
else:
    print('It is not hot.')
    print('Bring a jacket.')
print('Goodbye.')
```



Multi-way if Statement

```
def temperature(t):
    if t > 86:
        print('It is hot')
    elif t > 32:
        print('It is cool')
    else:
        print('It is freezing')
    print('Goodbye')
```



1) The value of `t` is 90.
2) The value of `t` is 50.
3) The value of `t` is 20.

Ordering of Conditions

What is the wrong with this re-implementation of `temperature()`?

```
def temperature2(t):
    if t > 32:
        print('It is cool')
    elif t > 86:
        print('It is hot')
    else: # t <= 32
        print('It is freezing')
    print('Goodbye')
```

```
def temperature(t):
    if 86 >= t > 32:
        print('It is cool')
    elif t > 86:
        print('It is hot')
    else: # t <= 32
        print('It is freezing')
    print('Goodbye')
```

The conditions must be mutually exclusive, either explicitly or implicitly

```
def temperature(t):
    if t > 86:
        print('It is hot')
    elif t > 32: # 86 >= t > 32
        print('It is cool')
    else: # t <= 32
        print('It is freezing')
    print('Goodbye')
```

Exercise (Notebook)

Write function `BMI()` that:

- takes as input a person's height (in inches) and weight (in pounds)
- computes the person's BMI and *prints* an assessment, as shown below

The function does not return anything.

The Body Mass Index is the value (weight * 703)/height^2 . Indexes below 18.5 or above 25.0 are assessed as underweight and overweight, respectively; indexes in between are considered normal.

```
def BMI(weight, height):
  'prints BMI report'

  bmi = weight*703/height**2

  if bmi < 18.5:
    print('Underweight')
  elif bmi < 25:
    print('Normal')
  else: # bmi >= 25
    print('Overweight')
```

```
>>> BMI(190, 75)
Normal
>>> BMI(140, 75)
Underweight
>>> BMI(240, 75)
Overweight
```

Iteration

The general format of a `for` loop statement is

```
for <variable> in <sequence>:
    <indented code block>
<non-indented code block>
```

`<indented code block>` is executed once for every item in `<sequence>`

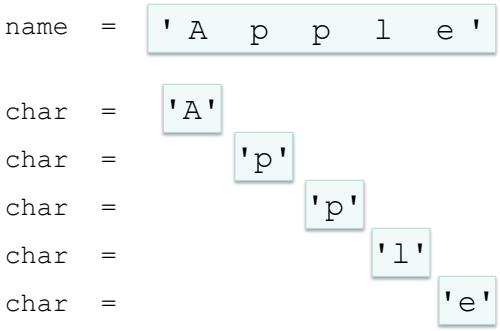
- If `<sequence>` is a string then the items are its characters (each of which is a one-character string)
- If `<sequence>` is a list then the items are the elements in the list

`<non-indented code block>` is executed after every item in `<sequence>` has been processed

There are different `for` loop usage patterns

Iteration Loop Pattern

Iterating over every item of an explicit sequence

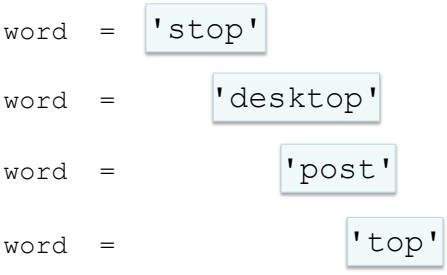



```
>>> name = 'Apple'
>>> for char in name:
        print(char)


A
p
p
l
e
```

Iteration loop pattern

Iterating over every item of an explicit sequence

```
for word in ['stop', 'desktop', 'post', 'top']:
    if 'top' in word:
        print(word)
```




```
>>>
stop
desktop
top
```

Iteration Loop Pattern

CLEMSON College of BUSINESS

Iterating over every item of an explicit sequence

- iterating over the characters of a text file

```
>>> infile = open('test.txt')
>>> content = infile.read()
>>> for char in content:
        print(char, end='')
```

- iterating over the lines of a text file

```
>>> infile = open('test.txt')
>>> lines = infile.readlines()
>>> for line in lines:
        print(line, end='')
```

Counter Loop Pattern

CLEMSON College of BUSINESS

Iterating over an implicit sequence of numbers

```
>>> n = 10
>>> for i in range(n):
        print(i, end=' ')


0 1 2 3 4 5 6 7 8 9
```

```
>>> for i in range(7, 100, 17):
        print(i, end=' ')


7 24 41 58 75 92
```

```
>>> for i in range(len('world')):
        print(i, end=' ')


0 1 2 3 4
```

This example illustrates the most important application of the counter loop pattern

Counter Loop Pattern

Iterating over an implicit sequence of numbers

```
>>> pets = ['cat', 'dog', 'fish', 'bird']
```

```
>>> for animal in pets:
        print(animal, end=' ')

cat dog fish bird
```

```
>>> for i in range(len(pets)):
        print(pets[i], end=' ')

cat dog fish bird
```

`animal =`	`'cat'`	`i =` `0`	`pets[0]` is printed
`animal =`	`'dog'`	`i =` `1`	`pets[1]` is printed
`animal =`	`'fish'`	`i =` `2`	`pets[2]` is printed
`animal =`	`'bird'`	`i =` `3`	`pets[3]` is printed

Counter Loop Pattern (Notebook)

Iterating over an implicit sequence of numbers… But why complicate things?

Let's develop function `checkSorted()` that:

- takes a list of comparable items as input
- returns True if the sequence is increasing, False otherwise

```
>>> checkSorted([2, 4, 6, 8, 10])
True
>>> checkSorted([2, 4, 6, 3, 10])
False
>>>
```

Implementation idea: check that adjacent pairs are correctly ordered

Exercise (Notebook)

Write function `arithmetic()` that:

- takes as input a list of numbers
- returns True if the numbers in the list form an arithmetic sequence, False otherwise

```
>>> arithmetic([3, 6, 9, 12, 15])
True
>>> arithmetic([3, 6, 9, 11, 14])
False
>>> arithmetic([3])
True
```

Accumulator Loop Pattern

Accumulating something in every loop iteration

For example: the sum of numbers in a list

```
>>> lst = [3, 2, 7, 1, 9]
>>> res = 0
>>> for num in lst:
        res += num
>>> res
22
```

shorthand notation

```
lst = [3, 2, 7, 1, 9]          res = 0

num = 3                        res = res + num   (= 3)
num =    2                     res = res + num   (= 5)
num =       7                  res = res + num   (= 12)
num =          1               res = res + num   (= 13)
num =             9            res = res + num   (= 22)
```

accumulator

Accumulator Loop Pattern

Accumulating something in every loop iteration

What if we wanted to obtain the product instead?
What should `res` be initialized to?

```
>>> lst = [3, 2, 7, 1, 9]
>>> res = 1
>>> for num in lst:
        res *= num
```

```
lst = [3, 2, 7, 1, 9]        res = 1

num = 3                      res *= num   (= 3)
num = 2                      res *= num   (= 6)
num = 7                      res *= num   (= 42)
num = 1                      res *= num   (= 42)
num = 9                      res *= num   (= 378)
```

Exercise (Notebook)

Write function factorial() that:

- takes a non-negative integer `n` as input
- returns `n!`

$$n! = n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times 3 \times 2 \times 1 \quad \text{if } n > 0$$
$$0! = 1$$

```
>>> factorial(0)
1
>>> factorial(1)
1
>>> factorial(3)
6
>>> factorial(6)
720
```

```
def factorial(n):
    'returns n! for input integer n'
    res = 1
    for i in range(2, n+1):
        res *= i
    return res
```

Exercise (Notebook)

Write function acronym() that:

- takes a phrase (i.e., a string) as input
- returns the acronym for the phrase

```
>>> acronym('Random access memory')
'RAM'
>>> acronym("GNU's not UNIX")
'GNU'
```

Exercise (Notebook)

Write function divisors() that:

- takes a positive integer `n` as input
- returns the list of positive divisors of `n`

```
>>> divisors(1)
[1]
>>> divisors(6)
[1, 2, 3, 6]
>>> divisors(11)
[1, 11]
```

ch5_codes.py

Nested Loop Pattern

Nesting a loop inside another loop

```
>>> n = 5
>>> nested(n)
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
```

```
>>> n = 5
>>> nested2(n)
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
```

When `j = 0` inner for loop should print `0`

When `j = 1` inner for loop should print `0 1`

When `j = 2` inner for loop should print `0 1 2`

When `j = 3` inner for loop should print `0 1 2 3`

When `j = 4` inner for loop should print `0 1 2 3 4`

```
def nested(n):
    for j in range(n):
        for i in range(n):
            print(i, end=' ')
        print()
```

```
def nested2(n):
    for j in range(n):
        for i in range(j+1):
            print(i, end=' ')
        print()
```

Exercise (Notebook)

Write function `inBoth()` that takes:

- 2 lists as input

and returns True if there is an item that is common to both lists and False otherwise

```
>>> inBoth([3, 2, 5, 4, 7], [9, 0, 1, 3])
True
>>> inBoth([2, 5, 4, 7], [9, 0, 1, 3])
False
```

Exercise (Notebook)

Write function `pairSum()` that takes as input:

- a list of numbers
- a target value

and prints the indexes of all pairs of values in the list that add up to the target value

```
>>> pairSum([7, 8, 5, 3, 4, 6], 11)
0 4
1 3
2 5
```

Two-dimensional Lists

The list `[3, 5, 7, 9]` can be viewed as a **1-D table**

`[3, 5, 7, 9]` =

3	5	7	9

How to represent a 2-D table?

		0	1	2	3
`[ [3, 5, 7, 9]`	=	0	3	5	7	9
`[0, 2, 1, 6]`	=	1	0	2	1	6
`[3, 8, 3, 1] ]`	=	2	3	8	3	1

A 2-D table is just a list of rows (i.e., 1-D tables)

```
>>> lst = [[3,5,7,9],
           [0,2,1,6],
           [3,8,3,1]]
>>> lst
[[3, 5, 7, 9],
[0, 2, 1, 6],
[3, 8, 3, 1]]
>>> lst[0]
[3, 5, 7, 9]
>>> lst[1]
[0, 2, 1, 6]
>>> lst[2]
[3, 8, 3, 1]
>>> lst[0][0]
3
>>> lst[1][2]
1
>>> lst[2][0]
3
>>>
```

Nested Loop Pattern and 2-D Lists (Notebook)

A nested loop is often needed to access all objects in a 2-D list

```
def print2D(t):
    'prints values in 2D list t as a 2D table'
    for row in t:
        for item in row
            print(item, end=' ')
        print()
```

(Using the iteration loop pattern)

```
>>> table = [[3, 5, 7, 9],
             [0, 2, 1, 6],
             [3, 8, 3, 1]]
>>> print2D(table)
3 5 7 9
0 2 1 6
3 8 3 1
>>> incr2D(t)
>>> print2D(t)
4 6 8 10
1 3 2 7
4 9 4 2
>>>
```

```
def incr2D(t):
    'increments each number in 2D list t'

    # for every row index i
        # for every column index j
            t[i][j] += 1
```

(Using the counter loop pattern)

Exercise (Notebook)

CLEMSON College of BUSINESS

Implement function pixels() that takes as input:

- a two-dimensional list of nonnegative integer entries (representing the values of pixels of an image)

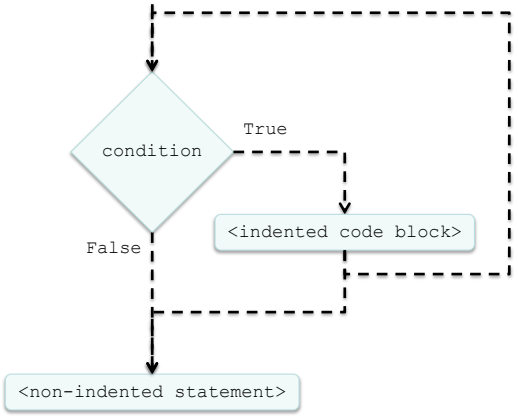
and returns the number of entries that are positive (i.e., the number of pixels that are not dark). Your function should work on two-dimensional lists of any size.

```
>>> lst = [[0, 156, 0, 0], [34, 0, 0, 0], [23, 123, 0, 34]]
>>> pixels(lst)
5
>>> lst = [[123, 56, 255], [34, 0, 0], [23, 123, 0], [3, 0, 0]]
>>> pixels(lst)
7
```

while Loop

```
if <condition>:
    <indented code block>
<non-indented statement>
```

```
while <condition>:
    <indented code block>
<non-indented statement>
```

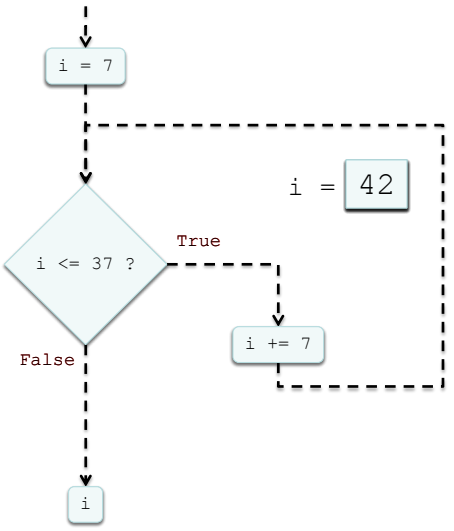


while Loop

Example: compute the smallest multiple of 7 greater than 37.

Idea: generate multiples of 7 until we get a number greater than 37

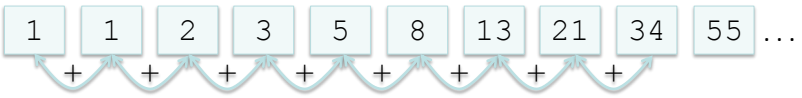
```
>>> i = 7
>>> while i <= 37:
        i += 7

>>> i
42
```



Sequence Loop Pattern (Notebook)

Generating a sequence that reaches the desired solution

Fibonacci sequence



Goal: the first Fibonnaci number greater than some `bound`

Infinite Loop Pattern (Notebook)

An infinite loop provides a continuous service

```
>>> hello2()
What is your name? Sam
Hello Sam
What is your name? Tim
Hello Tim
What is your name? Alex
Hello Alex
What is your name?
```

A greeting service

The server could instead be a time server, or a web server, or a mail server, or…

Loop-and-a-half Pattern

Cutting the last loop iteration "in half"

Example: a function that creates a list of cities entered by the user and returns it

The empty string is a "flag" that indicates the end of the input

```
>>> cities()
Enter city: Lisbon
Enter city: San Francisco
Enter city: Hong Kong
Enter city:
['Lisbon', 'San Francisco', 'Hong Kong']
>>>
```

```
def cities():
    lst = []

    city = input('Enter city: ')

    while city != '':
        lst.append(city)
        city = input('Enter city: ')

    return lst
```

```
def cities2():
    lst = []

    while True:
        city = input('Enter city: ')

        if city == '':
            return lst

        lst.append(city)
```

`break` and `continue` Statements

The `break` statement:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with ***the statement that follows the loop body.***

The `continue` statement:

- is used inside the body of a loop
- when executed, it interrupts the current iteration of the loop
- execution continues with ***next iteration of the loop***

In both cases, only the innermost loop is affected

```
>>> before0(table)
2 3

4 5 6
```

```
>>> table = [
        [2, 3, 0, 6],
        [0, 3, 4, 5],
        [4, 5, 6, 0]]
```

```
>>> ignore0(table)
2 3 6
3 4 5
4 5 6
```

```
def before0(table):
    for row in table:
        for num in row:
            if num == 0:
                break
            print(num, end=' ')
        print()
```

```
def ignore0(table):
    for row in table:
        for num in row:
            if num == 0:
                continue
            print(num, end=' ')
        print()
```

Exercise (Notebook)

Write function `negative()` that:

- takes a list of numbers as input
- returns the index of the first negative number in the list or -1 if there is no negative number in the list

```
>>> lst = [3, 1, -7, -4, 9, -2]
>>> negative(lst)
2
>>> negative([1, 2, 3])
-1
```

Exercise (Notebook)

Write function power() that:

- takes a positive integer ***n*** as input
- print 2**i for i = 1, 2, …, n
- return nothing

```
>>> power(2)
2 4
>>> power(3)
2 4 8
>>> power(10)
2 4 8 16 32 64 128 256 512 1024
```

Exercise (Notebook)

CLEMSON College of BUSINESS

Write function is_prime() that:

- takes a positive integer ***n*** as input
- returns True if ***n*** is a prime number; return False otherwise

[Hint] Divisors of a prime number ***p*** are only 1 and itself (***p***)

```
>>> is_prime(2)
True
>>> is_prime(6)
False
>>> is_prime(11)
True
```

Exercise (Notebook)

CLEMSON College of BUSINESS

Write function find_largest_prime() that:

- takes a positive integer ***n*** as input
- returns the largest prime number that is smaller than ***n***

[Hint] Use the is_prime() function we just implemented

Exercise (Notebook)

Write function bubbleSort() that:

- takes a list of numbers as input and sorts the list using BubbleSort

The function returns nothing

```
>>> lst = [3, 1, 7, 4, 9, 2, 5]
>>> bubblesort(lst)
>>> lst
[1, 2, 3, 4, 5, 7, 9]
```

Questions?