CS 17700

Logical Operators,
Strings,Lists and File

Week 4

Announcements

Project-1 is released. Due date is 9/27/2015 at 23.59 PM

Overview

- Logical Operators and Decision Structures
- Strings
- Lists
- File Operations

Logical Operators

Operator	Description	Example
==	Checks if the two values s are equal or not, if yes then condition becomes true.	a == b
!=	Checks if the two values are equal or not, if values are not equal condition becomes true	a != b
<	Checks if the left value is less than the right value. If yes then condition is true	a < b
>	Checks if the left value is greater than the right value. If yes then condition is true	a > b
<=	Checks if the left value is less than OR equal to right value. If yes then condition is true.	a <= b
>=	Checks if the left value is greater than OR equal to the right value. If yes then condition is true.	a >= b

Booleans

Boolean (logical) expressions:

An expression that can be evaluated as **True** or **False**

We use logical expression in everyday language:

If it is sunny today, then I should not need an umbrella.

Here *is it sunny today?* is a logical expression: its value can be either **True** or **False**

Code-like examples:

Assume **x=4**
x>3
Result-**True**

Assume **a_string="abc"**
type(a_string)==int
Result-**False**

Boolean (AND)

Examples:

Suppose a=**True**, b=**False**

a and b= ?

a and True=?

Suppose x=1, y=1

x > 0 and x <=2 –Result?

y > 0 and y >=3 –Result?

x	*y*	*x and y*
T	T	T
T	F	F
F	T	F
F	F
F

Booleans (OR)

Examples:-
Suppose a=**True**, b=**False**
a or b=?
a or True=?

Suppose x=1
x <= 0 or x > 2
Result?
x > 5 or x < 10
Result?

x	*y*	*x or y*
T	T	T
T	F	T
F	T	T
F	F	F

Boolean (NOT)

Examples:-

Suppose a=**True**, x=2

not a=?

not (not a)=?

not x > 3=?

DeMorgan's law:-

not (a or b) == (not a) and (not b)

not (a and b) == (not a) or (not b)

x	*not x*
T	F
F	T

Exercise

For what values of x, y, and z does the following statement evaluate to True?

not (z != 4 and z == 2) and not (y == 0 or x == 3)

A) x = 1, y = 0, z = 2
B) x = 3, y = 10, z = 12
C) x = 2, y = 1, z = 3
D) x = 3, y = 1, z = 4
E) x = 0, y = 0, z = 0

Exercise

For what values of x, y, and z does the following statement evaluate to True?

```
not ( z != 4 and z == 2 ) and not ( y == 0 or x == 3 )
```

Answer:
Option C: x = 2, y = 1, z = 3

Decision Structures

Why do we need Booleans?

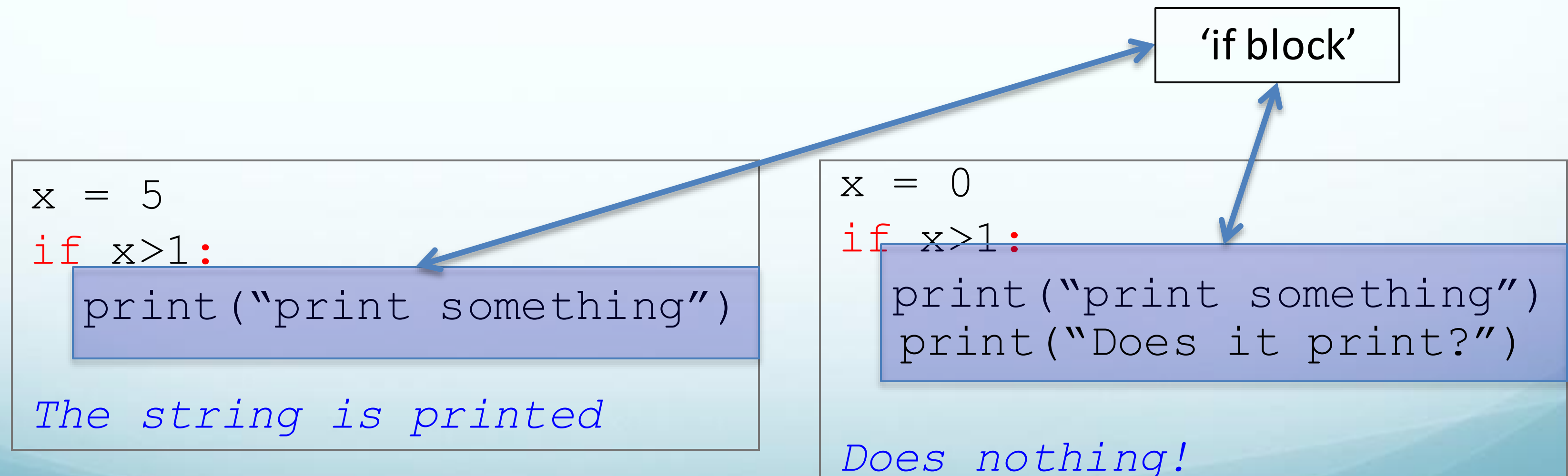
They can be used in decision structures

When used in the decision structure, the subsequent code will be executed only when the Boolean expression which is evaluated as the condition turns out to be 'True'

Decision Structures

If statement

An if statement takes a logical expression and evaluates it.
If it is True, the statements in the if block are executed
If it is False, they are not executed.
Simple decision examples



Decision Structures

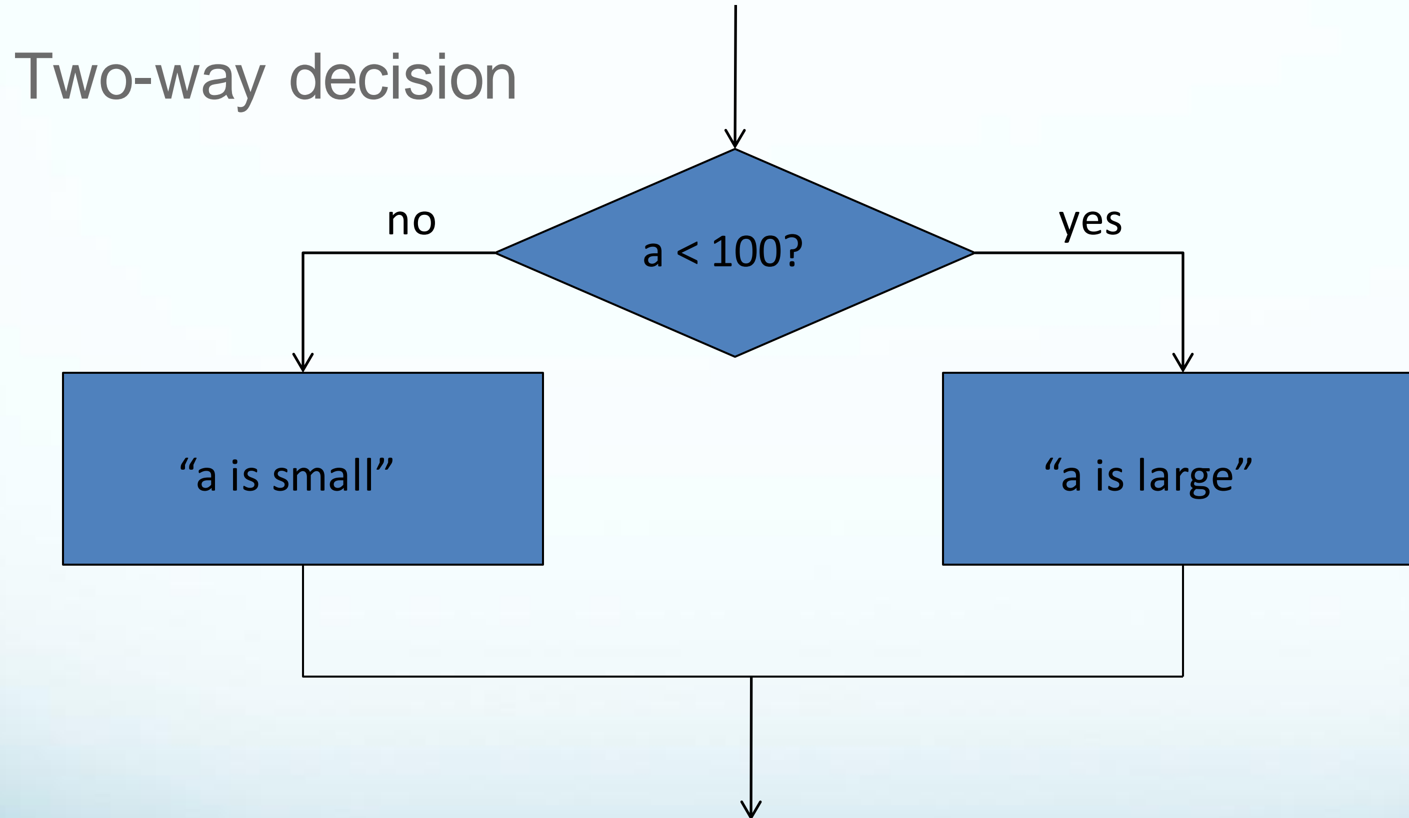
Two-way decisions are designated by if, else blocks
If the expression is True the if block is executed, otherwise the else block is executed

```
a = 45
if a < 100:
  print("a is small")
else:
  print("a is large")

>>> a is small
```

```
a = 153
if a < 100:
  print("a is small")
else:
  print("a is large")

>>> a is large
```

Decision Structures

Two-way decision



Decision Structures

Multi-way decision

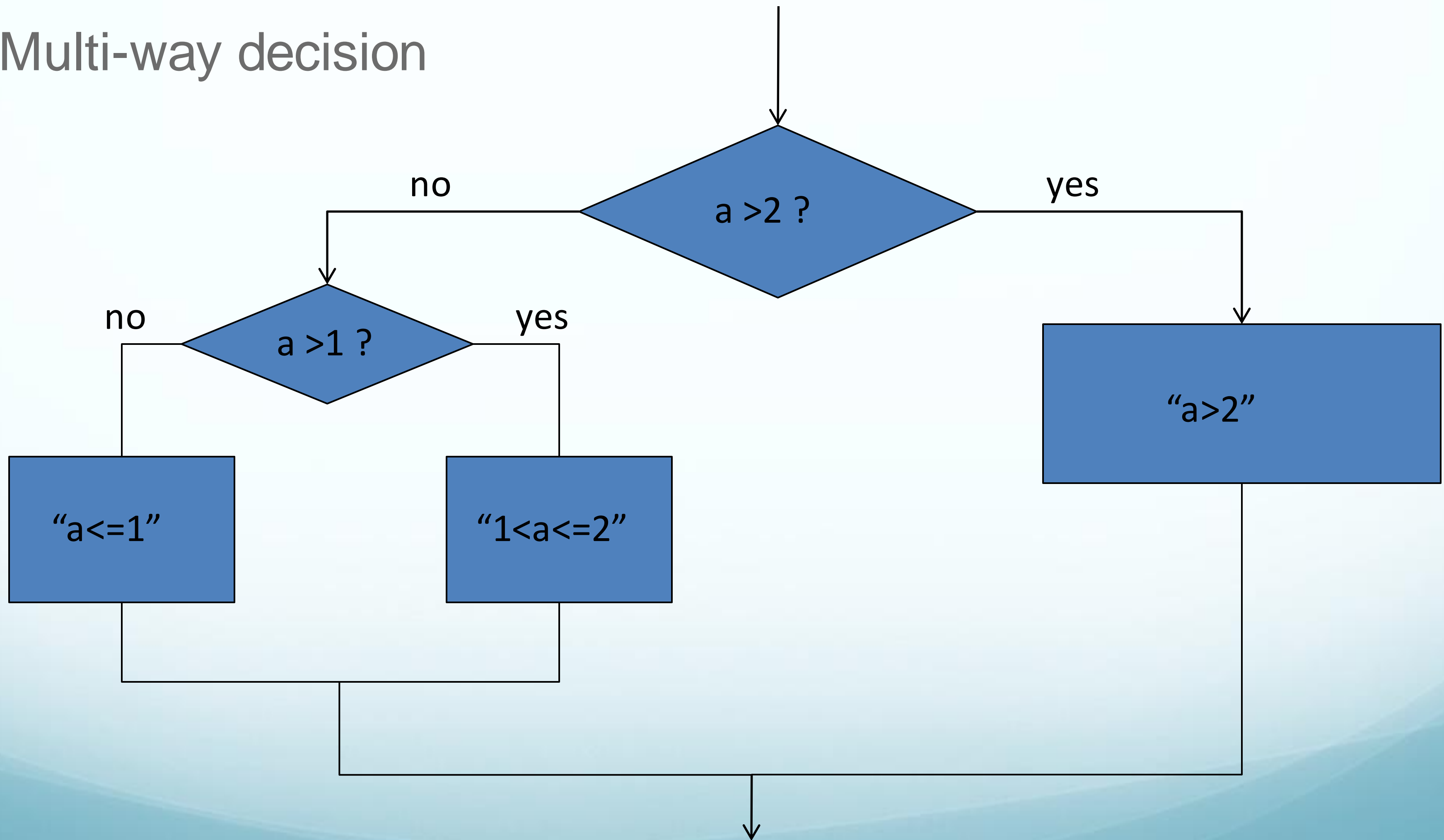
Decision statements can be nested within one another creating complex logic

```
a = 1.5
if a > 2:
   print("a>2")
else:
   if a > 1:
      print("1<a<=2")
   else:
      print("a<=1")

>>> 1<a<=2
```

```
a = 1.5
if a > 2:
   print("a>2")
elif a > 1:
   print("1<a<=2")
else:
   print("a<=1")

>>> 1<a<=2
```

Decision Structures

Multi-way decision



What will be printed?

```
credits=78
GPA=3.5
if credits >= 120 and GPA >=2.0:
        print('You are eligible to graduate!')
else:
        print('You are not eligible to graduate.')
```

Exercise

What is the output of the following code?

```
a=-5, b=0
if a>=0 and b>=0:
        print (“Both a and b are positive”)
elif a<0 and b>=0:
        print (“a is negative, b is positive”)
elif a<0 and b<0:
        print (“Both a and b are negative”)
else:
        print (“a is positive, b is negative”)
```

Strings

- Strings are amongst the most popular types in Python. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes.
- Creating strings is as simple as assigning a value to a variable. For example:

var1 = 'Hello World!'

var2 = "Python Programming"

Strings

Accessing Values in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring.

To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring

Strings

- Example:

```
var1 = ‘Hello World!’

var 2 = “Python Programming”

print(“var1[0] = “, var1[0])

print(“var2[1:5]  =“,var2[1:5])
```

Strings

- Example:

```
var1 = ‘Hello World!’
var 2 = “Python Programming”
print(“var1[0] = “, var1[0])
print(“var2[1:5]  =“,var2[1:5])
```

- This produces the output:

```
H
ytho
```

Strings

- Some String functions
- Capitalize() – First letter will be capitalized.
- Endswith() – Determines string or substring ends with suffix
- Lower() –converts to lower case letters
- Upper() –converts to upper case letters
- Find() –returns the index of a string or a substring

Strings

```
print ("hello world".capitalize())
Hello world
print ("Hello World".endswith("Hello"))
False
print ("HeLlO WoRlD".lower())
hello world
print ("Hello World".upper())
HELLO WORLD
print ("Hello World".find("World"))
6
```

Strings

- Escape Sequences
- In strings, a backslash \ character followed by one or more characters is used to represent any character that cannot be displayed in a string, such as the following:
- A character that does not appear on a standard keyboard;
- The single quote or double quote character that is being used to indicate the start and end of the string;
- The backslash character itself.

Strings

- Examples

```
var1 = 'doesn \'t'
>>> print(var1)
doesn 't
```

Strings

- Examples:

```
var1 = "new \none"
>>> print(var1)
new
one
```

Lists

- Multiple elements are stored consecutively in memory
- **Syntax:**

```
[elm0, elm1, elm2, …, elmn]
```

- Lists can be empty
  - `[]`
- Lists elements can be of different types
  - `[1, 2, ["ABCD", "EFG"], 3, 4]`
- Lists can be nested
  - `[[1,2], 4, [6, 10, [11, 12]], 5]`

Lists

- **Index:** provide us a quick mechanism for accessing a *given* element that is contained within a list
  - The index starts from 0, NOT from 1

[] Notation

- `a[i]` : gives a name to the *i*th element of a list
- `a = “Sally”`
  - `a[i]` is the i+1 character of Sally
  - In the example above, `a[2]` is the character ‘l’
- `a = list(range(0, 10))`
  - `a[i]` is the i+1 number in the range of 0 to 9
  - In the list above, `a[2]` is 2

Lists: Examples

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

Lets Make it More Concrete

```
a = 10
b = range(0,5)
c = "Sally"
```

	a	10
b[0]	b	0
b[1]		1
b[2]		2
b[3]		3
b[4]		4
c[0]	c	S
c[1]		a

....

Negative Indexes

- What happens if we use a negative index? Do we get an error?

```
x = list(range(10))
print(x[-1])                ← this will print 9
print(x[-10])               ← this will print 0
print(x[-11])               ← Error!
```

```
>>> print(x[-11])

Traceback (most recent call last):
  File "<pyshell#173>", line 1, in <module>
    print(x[-11])
IndexError: range object index out of range
```

- **Under the hood:**
  - **If you pass in a negative index, Python adds the length of the list to the index**

Lets Make it More Concrete

```
a = 10
b = range(0,5)
c = "Sally"
```

	a	10
b[-5]	b	0
b[-4]		1
b[-3]		2
b[-2]		3
b[-1]		4
c[-5]	c	S
c[-4]		a

….

More Complex Lists

- y is an example of a list. Each element is a string:

```
y = [“ABCD”, ”BCD”, ”CD”, ”D”]
```

- as you can see each element can be of different length
- The list elements can also be different types:

```
y = [“ABCD”, [1,2,3] , ”CD”, ”D”]
```

Indexing into Nested Lists

- Suppose we wanted to extract the value 3

```
y = ["ABCD", [1,2,3] , "CD", "D"]
y[1][2]
```

- The first set of **[]** get the element at position 1 of y. The second **[]** selects the element at position 2 of the element y. This is equiv. to:

```
z = y[1]
z[2]
```

Typical mistakes

- Undershooting the bounds
  - `a = “hello”` `a[-6]`
- Overshooting the bounds
  - `a = “hello”` `a[5]`
- Off by one
  - a[0] vs a[1]
  - By convention we use 0-based indexing

```
a=“hello”
print(a[0])
print(a[1])
```

Assigning to Lists

- The [] syntax not only allows us to retrieve the value of a given element, it also lets us change the content of that memory location
  - Namely, we can assign to that location

```
b=list(range(0,5))
b[2] = 100
print(b[2])
b[2] = b[2] - 40
print(b[2])
```

	a	10
b[0]	b	0
b[1]		1
b[2]		2
b[3]		3
b[4]		4

Assigning to Lists

- The [] syntax not only allows us to retrieve the value of a given element, it also lets us change the content of that memory location
  - Namely, we can assign to that location

```
b=list(range(0,5))
b[2] = 100
print(b[2])
b[2] = b[2] - 40
print(b[2])
```

	a	10
b[0]	b	0
b[1]		1
b[2]		100
b[3]		3
b[4]		4

Assigning to Lists

- The [] syntax not only allows us to retrieve the value of a given element, it also lets us change the content of that memory location
  - Namely, we can assign to that location

```
b=list(range(0,5))
b[2] = 100
print(b[2])
b[2] = b[2] - 40
print(b[2])
```

	a	10
b[0]	b	0
b[1]		1
b[2]		60
b[3]		3
b[4]		4

Operations on Lists

- len(): gives you the “length” or number of elements in a list.

```
len([0,1,2,3,4,5])
6
```

- Recall the example of printing each element of a string:

Operations on Lists

- append(): append an element at the end of a given list.

```
c = [1, 2, 3, 4, 5]
c.append(6)
```

  Results in c having an additional element:
  `[1, 2, 3, 4, 5, 6]`

- Just like we can *concatenate* strings we can concatenate lists

```
print ([1, 2, 3] + [4, 5, 6])
```

  - Will print: [1, 2, 3, 4, 5, 6]

- Just like we can *slice* strings we can also slice lists

```
b = [1, 2, 3, 4, 5, 6]
print (b[2:5])
```

  - Will print [3, 4, 5]

File Processing

- The process of *opening* a file involves associating a file on disk with an object in program memory.
- We can manipulate the file by manipulating this object.
  - Read from the file
  - Write to the file

File Processing

- For reading or writing a file, you need to **open** the file initially
- **open(filename, access_mode)** opens the filename
  - *Note*: if you do not provide a full path the file is assumed to be in the same directory where your program exists. If you are typing directly into IDLE the file must be where python is installed. (C:\Python32\)
- **access_mode** specifies the purpose of opening the file
  - "r" means *read ONLY*
  - "w" means *write ONLY, overwrites the file if the file exists*

Methods on Files

- <fileobject>.method() syntax: this time files are our object
  - file = open(“myfile”, “w”)
- file.read() -- reads the file as one string
- file.readline() - read the next line of the file as string
- file.readlines() - reads the file as a list of strings
  - read() and readlines() can only be used once without closing and reopening the file.
- file.write(data) - allows you to write to a file
- file.close() - closes a file

Extracting Data

- Data in files or web are useful and we are interested in extracting this information for use in our programs.
- Data in files or web are formatted as string data type.
- Therefore, we need to apply our knowledge in File I/O, Strings and Lists in order to extract these information.

Print File Contents

The read() operation:

```
def main():
    myfile = open(“students.txt”, “r”)
    content = myfile.read()
    print(content)
    myfile.close()

main()
```

```
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
```

students.txt

Print File Contents

The read() operation:

```
def main():
    myfile = open("students.txt", "r")
    content = myfile.read()
    print(content)
    myfile.close()

main()
```

Output:

```
>>>
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
>>>
```

```
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
```

students.txt

Print File Contents

The readlines() operation:

```
def main():
    myfile = open("students.txt", "r")
    content = myfile.readlines()
    print(content)
    myfile.close()

main()
```

```
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
```

students.txt

Print File Contents

The readlines() operation:

```
def main():
    myfile = open("students.txt", "r")
    content = myfile.readlines()
    print(content)
    myfile.close()

main()
```

```
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
```

students.txt

Output:

```
>>>
['Jim 75\n', 'Alice 95\n', 'Alex 75\n', 'Kate 75\n', 'John 95\n', 'Suzan 55\n',
'Tim 55\n', 'Sarah 85\n', 'Ann 95']
>>>
```

Print File Contents

The readline() operation:

```
def main():
    myfile = open("students.txt", "r")
    content = myfile.readline()
    print(content)
    myfile.close()

main()
```

```
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
```

students.txt

Print File Contents

The readline() operation:

```
def main():
    myfile = open("students.txt", "r")
    content = myfile.readline()
    print(content)
    myfile.close()

main()
```

```
Jim 75
Alice 95
Alex 75
Kate 75
John 95
Suzan 55
Tim 55
Sarah 85
Ann 95
```

students.txt

Output:

```
>>>
Jim 75

>>>
```

findAverage

- In this Example, the file students.txt contains the following data:

Jim 75
Alice 95
Alex 75
…

- We are interested in finding the average of the class

Finding Average of Grades

1. Open “students.txt” for reading
2. Read the file content into a string using read().
3. Extract grades
4. Calculate the average.
5. Print out the average

Important Note

- Data in files are stored as strings. If we are interested in the numeric value of the data, then we need to convert string values into numeric using functions: int or float
- Example:

```
>>> x = “150”
>>>xValue = int(x)
>>>y = “159.7895”
>>>yValue= float(y)
```

More String Manipulation

- **str.split(sep)**
  - Returns a **new** list of the words in the string, using *sep* as the delimiter string

```
“test1 test2 test3”.split(“ “) → ['test1','test2','test3']
```

```
str = "Line1\nLine2\nLine3”
print(str.split( ))
print(str.split('\n’))
print(str.split(‘\n’,1))
```

Output?

More String Manipulation

- **str.split(sep)**
  - Returns a **new** list of the words in the string, using *sep* as the delimiter string

```
"test1 test2 test3".split(" ") → ['test1','test2','test3']
```

```
str = "Line1\nLine2\nLine3"
print(str.split( ))
print(str.split('\n'))
print(str.split('\n',1)
```

Output-

['Line1', 'Line2', 'Line3']

['Line1', 'Line2', 'Line3']

['Line1', 'Line2\nLine3']

findAverage

```
def findAverage(fileName):
  file = open(fileName, "r")
  for line in file.readlines():
    #do work here!
```

- In each iteration of the for loop, the variable line will contain a line within the file. Remember in the previous slides that file.readlines creates a list, having as many elements as the lines in the file.

findAverage

- By observing the list content, we see that each string contains: name space score

```
def findAverage(fileName):
    file = open(fileName, 'r')
    Sum = 0
    Count = 0
    for line in file.readlines():
        sublist = line.split(' ')
        Sum = Sum + int(sublist[1])
        Count += 1
    print (Sum/Count)
    file.close()
```

ANY QUESTIONS?